Aalto University
School of Electrical
Engineering

Resource Management in Container-based Mobile Edge Computing

Muhammad Farooq Tufail

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology
Espoo, August 14, 2018

Supervisor

Professor Yu Xiao, Aalto University

Advisor

Marius Noreikis M.Sc. (Tech.)

Aalto University
School of Electrical
Engineering

Abstract of Master's Thesis

Author: Muhammad Farooq Tufail

Title:
Resource Management in Container-based Mobile Edge Computing

Date: August 14, 2018 **Pages:** vii + 67

Major: Communications Engineering **Code:** ELEC3029

Supervisor: Professor Yu Xiao

Advisor: Marius Noreikis M.Sc. (Tech.)

Mobile edge computing is a promising technology which provides support to time-sensitive applications by pushing centralized cloud processing capabilities to distributed Fog nodes. These fog nodes are deployed at one-hop distance from end-user and provide real-time data processing capabilities at the edge of network. Due to service provisioning at the edge of network, no congestion occurs at the core of network, quality of service (QoS) is improved and the overall network operational cost is significantly reduced. However, these nodes have limited capabilities such as processing, storage and coverage so, they face challenge of mobility support for a mobile user when continued service (i.e. zero downtime) is required during handovers between edge nodes. Furthermore, they also need an effective task allocation and resource management strategy to ensure smooth operation of edge services. Unlike traditional VM based environment in Fog Computing, this work explores lightweight Docker containers to deploy and migrate services. In this work, an interactive event-driven dashboard is developed for real-time edge node registration, system monitoring, service initiation and migration. Then, motivated by Fog Following Me [47], a couple of resource allocation schemes (i.e. algorithm-I & II) have been introduced to dynamically manage the compute resources among fog nodes. For smooth service operation and stable migration, an application profiling feature has been introduced which assigns the needed quota for an application requirement in terms of CPU, GPU and RAM. The developed system′s performance is evaluated by conducting various experiments. The experimental results clearly demonstrate and verify the working feasibility of the whole system′s operation in context of edge computing. However, the observed processing delays during service migration marks the limitation of Docker and suggest the need to use latest optimization tools to cut down the network delays and ensure zero-downtime service migration.

Keywords: Internet of Things, Fog Computing, CRIU, Docker Containers

Language: English

Acknowledgements

By the grace of Almighty Allah, I have been able to write this thesis.

At first, I would like to express my gratitude towards Professor Yu Xiao, who granted trust on me and provided mean opportunity to explore and work on this project. I am thankful for her valuable time, generous support and more importantly knowledge she shared in brainstorming sessions and weekly meetings.

Secondly, I want to thank my advisor, Marius who provided me weekly mentoring sessions and guided me in the right direction. Through his feedback and guidance, I have been able to complete the implementation phase and resolve all technical and design issues. I also want to thank CRIU project community who addressed my queries and shared their time and knowledge with me. Thanks to my friends Dani, Asad, Waqas, Hassaan and Abdullah for their moral support and memorable time.

Then, I would like to thank my parents and family who supported me at every time and kept my spirits high. Without their prayers and support, I wouldn't have been able to come to Finland for higher education. Finally, I would like to thank Aalto University and people of Finland for providing me best environment to learn from this high-ranked institute.

Espoo, August 14, 2018

Muhammad Farooq Tufail

Abbreviations and Acronyms

VM	Virtual Machine
IoT	Internet of Things
CRIU	Checkpoint and Restore In User Space
FN	Fog Node
5G	5th Generation for Mobile Communication
MEC	Mobile Edge Computing
MCC	Mobile Cloud Computing
IP	Internet Protocol
MM	Mobility Management
RM	Resource Management
AR	Augmented Reality
CPU	Central Processing Unit
GPU	Graphics Processing Unit
RAM	Random Access Memory
IaaS	Infrastructure as a Service
WLAN	Wireless Local Area Network
WIFI	Wireless Fidelity
RSSI	Received Signal Strength Indicator
UCP	Universal Control Plane
PaaS	Platform as a Service
VMM	Virtual Machine manager
OS	Operating System
cgroups	Control Groups
IPC	Inter-Process Communication
MAC	Mandatory Access Control
SE Linux	Security Enhanced Linux
MCS	Multi-Category Security
HW	Hardware
LMCTFY	Let Me Contain That For You
CI/CD	Continuous Integration and Development

REST	Representational state transfer
API	Application Programmable Interface
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
DMTCP	Distributed Multi-Threaded Checkpoint
I/O	Input Output
HPC	High Performance Computing
SDN	Software Defined Networking
DSO	Data Service Operators
ADSS	Authorized Data Service Subscribers
MQTT	Message Queuing Telemetry Transport
MTCP	Multipath TCP
NFS	Network File System
MVC	Model View Controller
URL	Uniform Resource Locator

Contents

Chapter 1

Introduction

Ever since digital revolution, the internet technology has facilitated every aspect of human life. At present, about more than 54% of world population is connected to internet and this number increases significantly on daily basis [35]. A recent survey by Gartner [28] shows that by the end of 2020, more than 20.4 billion devices would be connected to internet. Thus, future internet is perceived to consists of complex communication scenarios in which every gadget and device would be connected to internet and it can reach out to any other device from internet. This paradigm is also known as Internet of Things (IoT). The key idea is to access information everywhere and at all time by integrating physical entities with virtual world. For this purpose, IoT devices are intelligently programmed to collect the data and then forward it to external systems or data centers for data analysis, computations and deciding on further actions [37].

The traditional two-tier model of IoT comprises of end-devices and remote cloud servers. These end-devices lack sufficient storage and processing capabilities. Thus, they heavily rely on remote cloud datacenters to perform various data-intensive tasks including speech recognition, augmented reality and computer vision etc. By offloading these computation-intensive tasks to cloud servers, the resource-poor end-devices are empowered by resource-rich computations in cloud and this ultimately facilitates the end-user. This model of IoT involving remote cloud is known as mobile cloud computing (MCC) [30]. However, MCC entails high end to end delays when service is provisioned from far located cloud servers. Therefore, it is not ideal for real-time applications which require low-latency computation response and fast processing e.g. real-time video processing, augmented reality (AR) based object

recognition, gesture recognition etc [47].

To address this low-latency (i.e. 1ms-10ms) requirement, a new concept is introduced which distribute and deploy cloud computing resources near to user and at the edge of network. This concept is referred to as mobile edge computing (MEC) or Fog Computing. As shown in Figure 1.1, it consists of 3-tier model: cloud, edge node and end-user.

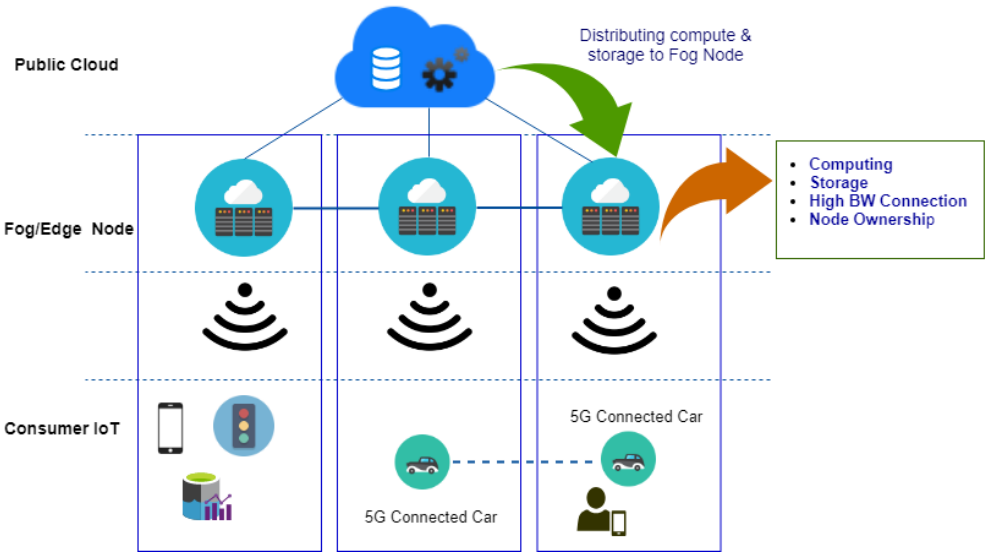

Figure 1.1: Moving From Centralized cloud to Distributed Cloudlets (Adapted from [11])

In this architecture, services are processed (near vicinity of end-user) at edge of network by edge nodes which are also termed as cloudlets or Fog Nodes. Fog nodes can be imagined as the distributed cloud servers or *data-centers in a box* which are equipped with sufficient compute resources (i.e. CPU, GPU and RAM). Usually, these nodes are placed at one-hop distance from end-user and due to their high bandwidth, they are fully capable to perform time-critical operations such as compute, control, decision making and communications etc. They exist in soft-state (i.e. IaaS) and possess self-managing capabilities without requiring any hardware management after installation. On the back-end, they are connected to cloud via a high bandwidth fixed or WLAN interface, so end-to-end delay is greatly minimized for local service provisioning.

In Fog & Mobile Edge Computing, the main problem is to dynamically manage compute resources and initiate user-applications without overloading the corresponding cloudlets. Another important challenge is service provisioning to a 5G mobile user (and autonomous connected cars), which moves at faster speeds and still wants uninterrupted service from the nearest fog node. To solve these issues, a concept known as **Fog Following Me** [47] is introduced. This concept presents an efficient task allocation strategy, which helps both stationary and mobile fog nodes to achieve the desired service latency. Whenever a mobile user leaves the coverage area of one fog node and enters the coverage area of another fog node, its corresponding service is migrated with zero down-time from source to destination. This feature of service migration not only provides continued service to mobile user but also gives possibility to offload services from one fog node to another. As a result, in case of congestion on a fog node, it′s all active services are migrated to another available fog node to ensure smooth operation of services and thus, avoiding un-necessary delays and quality loss issues.

At present, the traditional architecture of cloudlets is marked by VM (virtual machine) based solutions which may take minutes and hours to start an application [20]. Therefore, VM based cloudlets can undermine the advantages of Fog computing due to their high response time. The ideal solution in this case must be a lightweight implementation which can deploy and migrate services in edge cloud with the blink of an eye. Fortunately, this solution can be imagined with lightweight virtual machines or more specifically the Linux containers which only require application-specific binaries and files to initiate an application. Unlike VMs, they are small-sized and possess ability to reuse the host operating system. As a result, the deployment time of applications is greatly reduced to seconds.

1.1 Scope & Objectives

This research work aims to develop a resource management platform for container based edge cloud devices. The architecture of platform must be event-driven to get fine-grained control on all devices and abstract tasks based on defining certain events. The complete picture of platform can be imagined by defining the following research objectives:

- **Registration & Edge Node Discovery**
  The developed platform must support edge node discovery and registration of all edge nodes to its database. At real-time, each edge node will report its system utilization information including CPU, GPU and RAM to resource manager so edge node can be monitored for deploying and managing applications.

- **Application Profiling**
  To control and limit an application′s access to system resources, each application must be assigned a maximum quota in terms of CPU, GPU and RAM. This application profiling would also be consulted during service migration from one edge node to another.

- **Dashboard for Control Monitoring**
  This platform must be user-friendly and supports web-based user interfaces for configuring and launching containers. From its dashboard, a user can view and monitor the system at real time, deploy applications and view the system utilization at application level.

- **Support of Service Migration**
  The important feature of platform includes the support of service migration. This could be scheduled and triggered either by monitoring WIFI-RSSI of edge node or its system level utilization information.

- **Management of task allocation strategies**
  Another objective of this work includes the implementation of some service migration and offloading strategies. The idea is to make this platform flexible enough to accommodate multiple different task allocation strategies.

1.2 Contribution

Some important contributions of this work are as follows:

- Designing an event-driven platform for resource management across edge nodes.

- Exploring feasibility of Docker as a tool for container management (i.e. service provision and migration) in context of Fog Computing.

- Developing and testing a couple of strategies to migrate containers from one edge node to another.

1.3 Structure

The Thesis structure is distributed among following chapters:

Chapter 2 covers brief introduction about cloud virtualization technologies (i.e. containers and VMs) followed by focus on famous container management tool known as Docker.

Chapter 3 briefly summarizes relevant projects developed for resource management and container migration in Edge Computing. It compares and discusses various methods of resource allocation from both domains (i.e. MCC and MEC).

Chapter 4 presents the complete architecture of resource manager and explains its constituent technologies and framework. Docker′s experimental feature **docker-checkpoint** is introduced and discussed how it is used with CRIU (checkpint & restore in user space) to checkpoint and restore a docker container. In addition, this chapter also describes the working flow of each process from node discovery to service management. At the end, a couple of proposed migration strategies (i.e. algorithm-I & II) have been discussed which trigger container migration across the network.

Chapter 5 evaluates the developed platform and its features. For this purpose, some experiments are performed by emulating certain situations and observing system response and obtained results to test the feasibility of Docker as a tool for container orchestration. This chapter also does comparative analysis of developed system & its integration with open source container orchestration tools (such as Kubernetes, Docker Swarm).

Based on experiment results, Chapter 6 concludes by addressing major findings and drawbacks with current setup. Finally it formulates some guidelines for improvement so our system can become an ideal candidate for resource management in Fog Computing.

Chapter 2

Cloud Virtualization

Virtualization is the prevalent and promising technology introduced in mobile cloud computing. As a result, nowadays we have freedom of on-demand, multi-tenant and flexible deployments from public cloud hosting companies whether it requires Infrastructure as a service (IaaS) or platform as a service (PaaS). In this domain, some major cloud hosting companies include VMware, Citrix, Microsoft Azure and Google. The idea behind virtualization is to split up the resources of a single physical server into multiple logical and isolated instances (also termed as virtual machines). The added benefits include on-demand and flexible deployments, hardware independence, service isolation, system scalability and secure execution environments. These benefits help cloud providers to reduce deployment cost and instantiate services as they are needed.

2.1 Virtualization Technologies

At present, virtualization can be classified into two main categories: hypervisor-based virtualization and container-based virtualization.

2.1.1 Hypervisor-based Virtualization

In this type of virtualization, virtual machines are controlled and managed by a piece of software called as hypervisor or virtual machine manager (VMM). This software sits directly on top of host OS (Linux in this case) and abstracts virtual hardware for guest operating systems. In this way a full OS is installed for each virtual machine on top of this virtualized hardware. Some of its salient features include the

following:

- **Application Transparency:** The guest VMs and applications offer similar benefits as compared to when they run in real environment i.e. they are unaware of the fact being executed in a virtualized environment. Due to this reason applications and programs produce same results despite running as virtualized process.
- **VM Isolation:** Hypervisor-based deployed VMs are independent to each other and also separated from host system. In this way, applications belonging to one VM are unable to access applications of another VM instance. This VM-level isolation ensures that application failure within a VM affects only that specific VM instance and all other VMs and host OS are sandboxed.
- **VM Migration:** VM operating system and application-specific files reside completely on a virtual disk file. In case of system failure or in need of dynamic service provision, the virtual disk file can be easily migrated to remote server and then restart the VM to rapidly initiate the desired service.

However, this virtualization incurs large overhead during hardware emulation. Furthermore, installing a separate OS for each VM makes the final image size substantially higher. Therefore, VM′s take minutes to boot-up the whole system and start an application.

2.1.2 Container-based virtualization

Like VM′s, containers also aim to isolate an application with the possibility to move the binary artifacts between multiple hosts. However, they appear as lightweight VM′s with respect to their underlying architecture as shown in Figure 2.1.

Unlike HW virtualization in case of VM′s, containers utilize OS-level virtualization i.e. they share the host OS resources (OS kernel and run time environment) and run on top of a container driver engine. Compared with VM′s, they only need basic application-specific binaries and libraries to start an application without requiring a whole new version of OS. This reduces the base image size excessively and deployed container requires a relatively low hardware footprint when compared to VM based architecture. To perform service isolation and resource management among container processes, following Linux Kernel features are utilized:

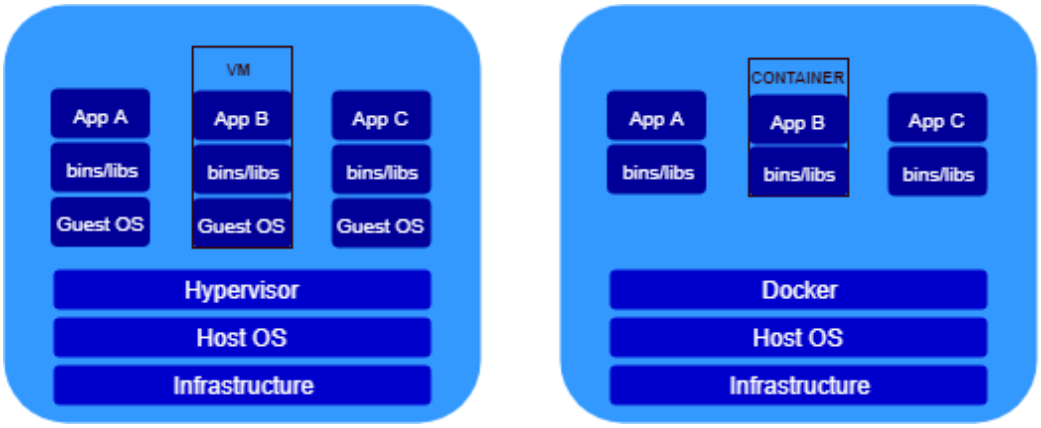

Figure 2.1: Comparison between VM and containers

1. **Control Groups (cgroups):** As the name suggests, control groups perform control operations such as resource allocation, prioritization, allow or deny feature and monitoring among group of user-defined processes [16]. They sub-divide various processes and their sub-processes into classified groups to restrict their resource consumption. This keeps a container's processes from utilizing too-much system resources.

2. **Linux Namespaces:** According to [17, 19, 23],Linux namespaces abstracts process isolation through kernel-restricted user-space views. These views limit the access of all namespace processes to global system resources in such a way that they can only view system resources belonging to their own namespace. The resources which can be assigned to a namespace include mountpoints, network devices, IPC, host and domain information etc. Each single container is mapped to a unique namespace which stays inaccessible to all other containers. In this manner, various running processes are compartmentalized and remain isolated from each other thus, enforcing container-to-container isolation.

3. **SE Linux:** SE Linux introduces a MAC (mandatory access control) mechanism to provide horizontal as well as vertical isolation. In horizontal isolation, it prohibits one container to access another one whereas in vertical isolation, it protects host operating system from all spawned containers. Due to SE Linux enforcement, container processes have limited access to system resources. Furthermore, this enforcement also implements a secu-

rity mechanism known as MCS (multi-category security) to prevent container′s interaction with each other.

Compared to hypervisor-based VM's, containers achieve isolation in an optimal manner. Therefore, they appear to be more advantageous. As mentioned in [2], some of the major gains involved with containers include the following:

- **Small HW Footprint:** Containers achieve environment isolation by using host OS features (i.e. cgroups and namespaces). This method reduces the CPU utilization and memory overhead of host OS as compared to hypervisor-based virtualization.
- **Process Isolation:** As every container is sandboxed and isolated from each other, so an update or crash to system libraries of one container affects only that specific container application and all other containers remain safe from that un-needed upgrade.
- **Fast Deployment:** Containers get spawned in a few seconds without requiring the need to install and restart OS. Similarly, an application upgrade involves only restarting the container without affecting host OS services.
- **Support of Multiple environments:** In conventional deployments involving a single host, an application supports only certain environment and may break out in other incompatible environments. However, containers avoid this conflicted environment execution as now every container is setup using the same base image hence, mitigating the chance of any differences in run-time environment.
- **Re-usability:** Once containers are spawned, they provide flexibility to re-use them by many similar applications without requiring a new OS install each time.
- **Support for micro-services:** Containers make it easy to deploy lightweight and agile services along with deployment support in multiple environment which ultimately boost the micro-services driven architecture and their development use cases.

2.2 Docker

In context of Fog computing, many candidates and container management platforms emerged for containerization such as LXC, coreOS, LMCTFY, openvz, vserver, Apache Mesos, runc and Docker etc. However, since past few years, Docker has become the only viable option and industry standard for OS-level virtualization on Linux. Developed by dot Cloud, Docker is an open-source project for system administrators and developers to create, migrate and deploy distributed cloud applications. On top of Linux standard features (i.e. namespaces, cgroups, SE Linux), it gives freedom to create loosely-spaced isolated user space environments which are termed as containers. It streamlines the product development life-cycle by giving flexibility of local deployment and testing and finally integrating it into production environments. Therefore, it is perfect for use-cases such as continuous integration and continuous development (CI/CD).

2.2.1 Docker Architecture

It follows a client-server architecture as demonstrated in Figure 2.2. A command-line tool (also known as docker) communicates with the server through a RESTFUL API to orchestrate the containerization process. On the server side, a docker daemon (dockerd) sits on the host OS to handle all client requests and is responsible to build, upload and download container images. In addition, it also creates and manages major objects of Docker such as docker networks, images, volumes and containers.

Whenever a docker command is executed on client-side, it is forwarded to docker daemon to perform necessary operation. For example, as shown in Figure 2.2, if docker pull nginx request is received from client, the server will fetch the nginx image from public repository (such as DockerHub) and store it on local host repository. Now to start a container with downloaded image, docker run command is made to server. At first, server will search in its local repository for the desired image (i.e. nginx) and then use it to start a container. However, if another request is received to start a nodejs application using node image. Then, the server finds no corresponding image in local repository, so it makes another pull request from public registry and download the necessary node base image. Finally, docker run command starts the container from downloaded image.

As docker provides flexibility to build custom images so for this purpose, docker build command is executed. In this process, docker adds more layers on top of base image, so that an extended image can be formed and pushed to public registry for private or public use. Accord-

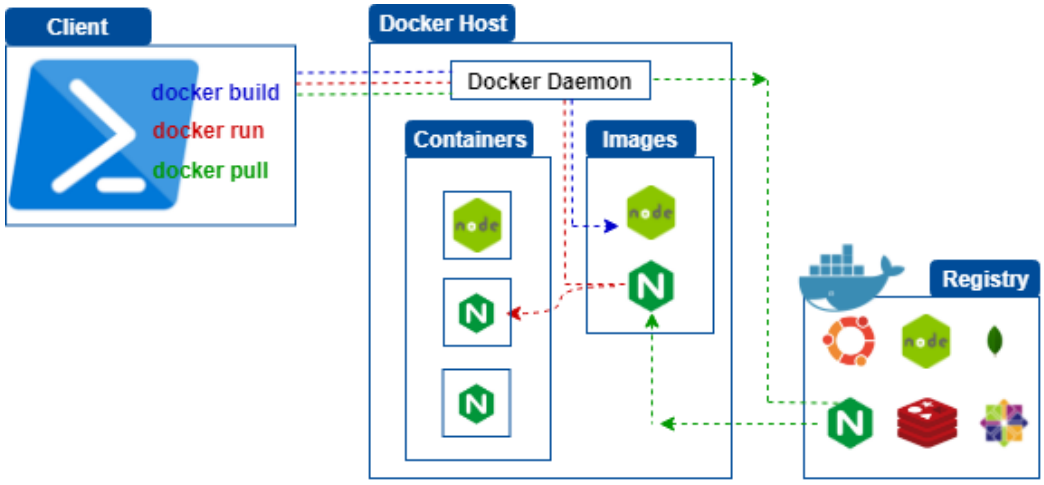

Figure 2.2: Docker Architecture (adapted from [1])

ing to Figure 2.2, the overall docker system consists of following three core elements:

1. **Docker Images**
   Docker images can be called as collection of read-only packages or templates [8] which encapsulate everything needed to start a containerized application. These templates contain application-specific code, configuration files, libraries, environment variables and required runtime environment. Docker daemon utilize these images to start any type of container. As shown in Figure 2.3, a docker image file comprises of a stack of multiple application layers or dependencies. However, Docker maintains a UnionFS (filesystem) which makes this layered model of an image to appear as a single virtual file system. Whenever a new file is added to a docker image or an existing file is modified , a new image layer is formed on top of earlier layers resulting into a new image. During this process, only those layers are rebuilt which have been changed keeping all other layers at their initial state. As a result, we get lightweight images which are small and fast-to-deploy.

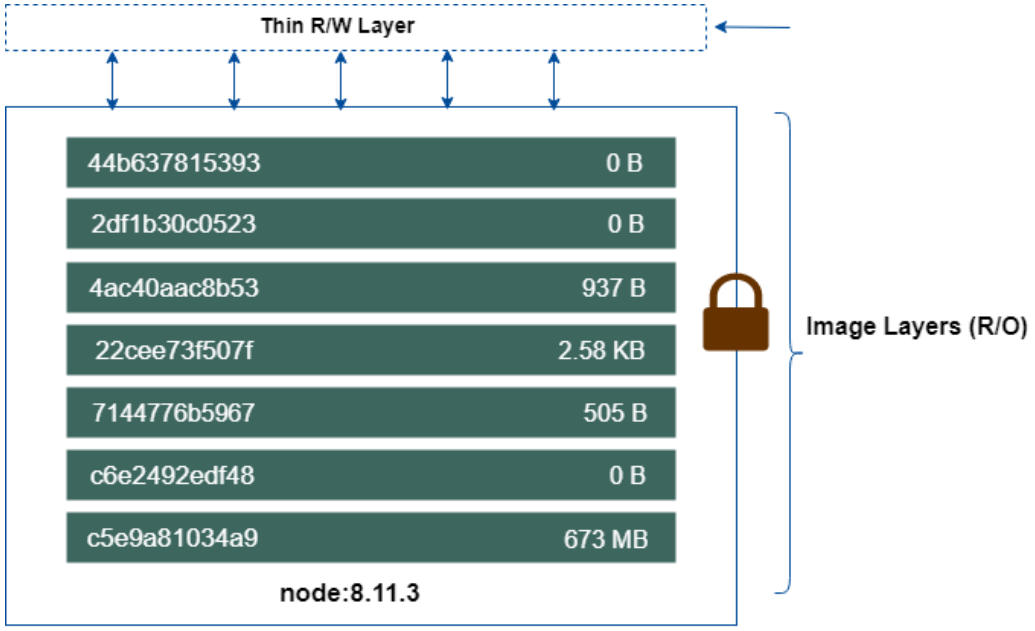

Figure 2.3: Container based on node:8 image (adapted from [1])

In general, two methods have been introduced to build a docker image:

- **Using Docker Containers**
  This is the simplest and easiest approach to create docker images on top of running instance of a docker container. At first, any changes made to configuration file or docker runtime result into a new read/write layer. Then, this extra layer is saved on top of existing image to publish a new image. However, this approach is inefficient because it also adds some extra layers from un-needed and temporary log files and resulting image becomes much larger than the original image size.
- **Dockerfile**
  A text-formatted file which includes step by step instructions and commands to build a docker image starting from a base image. Building an image using Dockerfile is the recommended approach because it avoids addition of unnecessary files and limit them to each layer. Within a Dockerfile, every instruction forms a new layer on base image in the specified order to create the final image. As shown in Figure

2.3, whenever a container is started, a thin writable layer known as "container-layer" is created on top of all underlying layers. In this approach, each new change to container′s run-time environment is stored in this thin R-W (read-write) layer. This layer also marks the difference between a docker image and a container. It exists until and unless a container is running, otherwise deleting a container also removes this writable layer but without affecting image layers. For demonstration purposes, a sample Dockerfile to build a nodejs counter application is shown below.

```
1        FROM node:8
2        WORKDIR /usr/src/app
3        COPY package*.json ./
4        RUN npm install
5        COPY . .
6        EXPOSE 8080
7        CMD ["npm","start"]
```

Listing 2.1: A Sample Dockerfile for nodejs Counter Application

It starts with FROM command which defines node:8 as base image for application. The second command WORKDIR specifies the working directory of container. On third line, nodejs application package JSON file is added from Docker client directory to image directory. Similarly, RUN command makes it possible to run the input commands directly in container′s shell. An EXPOSE instruction makes the specific network port (TCP/UDP) of container available to host. For example, in given Dockerfile, the tcp port 8080 is exposed externally so that container will listen on this port at run-time. However, docker avoids direct publishing of this port until and unless a container gets started by specifying the publish port with "-p" flag. Finally, CMD parameter shows the commands (i.e. npm start) which need to be executed immediately after starting a container.

2. Docker Registries

Docker manages a public repository known as registry to store images. According to [13], Docker Cloud and Docker Hub are two well-known public repositories which contain multiple images (including official releases such as nginx, nodejs, CentOS etc.) avail-

able to download for docker community. In addition, docker build command facilitates the docker community to create custom images and then, push them to Docker Hub for public or private use.

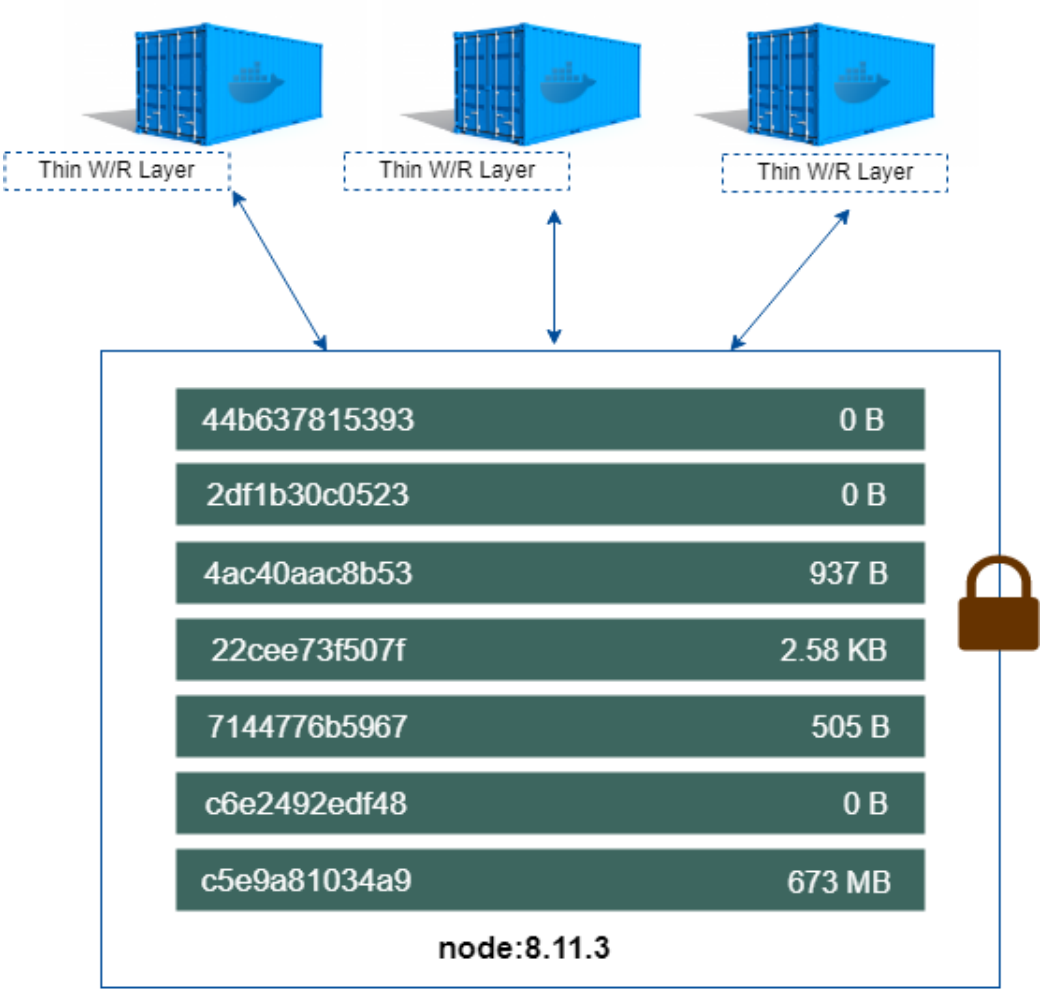

Figure 2.4: Docker Containers sharing same node Image (adapted from [1])

3. **Docker Containers**
   Docker containers can be defined as isolated Linux processes created by executing run time instances of docker images. These user-space segregated processes keep applications separate from

one another. Sitting on top of Linux host kernel, all containers share the same operating system and have native access to global system′s resources. Due to immutable property of docker images, we can create multiple instances of containers from same image and each container maintains its own data state. Thus, each running container creates its own writable layer on top of base image as shown in Figure 2.4. To store this writable layer as persistent data, docker volumes have been introduced which persist the container data directly on local file-system. However, to keep un-persistent data in writable layer of containers, docker storage drivers such as btrfs, aufs, zfs, vfs and overlay2 are used. These drivers manage local disks through UnionFS (union-filesystem) but at cost of performance loss (i.e. low read and write speed) when compared to docker volumes.

2.3 Checkpoint & Restore In User-Space (CRIU)

To realize the full potential of Fog Computing, container-based service migration has been introduced. The technology developed for this purpose is named as checkpoint-restore. The basic concept of this technology is to freeze a running container, save its memory state as a collection of image files and then restore the container from those image files on the destination. In past, many tools and projects have been developed to checkpoint and restore containers such as DMTCP (distributed multi-threaded check-pointing), BLCR (Berkeley Lab Checkpoint-Restart), PinPlay, OpenVZ and CRIU etc [5]. With respect to their checkpoint-restore implementation, all these projects fall into two main categories: low-level implementation (i.e. accessing Linux kernel for checkpoint-restore) and a high-level implementation (freezing applications in user-space environment). As low-level implementation demands more requirements on applications to be check-pointed (including the need of pre-compiling or loading special libraries) as well as need of complex modifications to Linux Kernel. Therefore, Linux community recommended a relatively more transparent and feasible method to achieve checkpoint and restore feature entirely in user-space [12]. Based on this recommendation, CRIU appears as a viable tool which exploits user-space to make snapshots of containerized applications and then restart them at a later stage from those check-pointed image files. In

this process, it requires no need to pre-load custom libraries because Linux Kernel (as of Kernel version 3.11) provides native support for all CRIU features.

2.3.1 CRIU Working Principle

Developed by Virtuozzo as an open-source project, CRIU at present is integrated into various container-based environments such as LXC, LXD, Docker, runc and OpenVZ etc. However, this work explores checkpoint and restore feature of CRIU using Docker. The overall checkpoint and restore feature can be divided into two main steps:

1. **Checkpoint**
   The main purpose of checkpoint is to collect the complete state of all running processes so that at later stage, this state information can be recreated for restarting the process. According to [4] , the overall checkpoint process is further sub-divided into following steps:

   - At first step, the process tree (including parent tree and all children processes) information of container is retrieved using /proc file-system to freeze the parent process and all its sub-processes.
   - In the second step, entire state information of application is collected and dumped in the form of image files on local disk. This state information includes many things such as file-descriptors, socket-pipes, network connections, cpu register, credentials, uids, gids, memory-mappings and timers etc.
   - With the help of process trace (i.e. ptrace debugging) interface, CRIU takes charge of whole process and terminates it.
   - Finally, CRIU utilizes that ptrace to inject a certain code known as "parasitic code" inside the stopped process. This code is executed directly from address-space of process to get access to its memory and then to save all its contents.
   - When all memory information is collected and written to disk storage, then process can be aborted or continue working if fault-tolerance is desired.

2. **Restore**
   It is the reverse-process of checkpoint because now CRIU will

have to recreate the processes from those image files which were check-pointed earlier. This restore process consists of following steps:

- First step involves reading image files and figure out which resources are shared by which processes and what is the correct order of restoring. Then, one process is enough to restore shared resources and another one is inherited for each remaining resource. For example, if one open file descriptor is shared by multiple sub-processes, so it is just opened once for all of them and then inherited for others.
- In the second step, CRIU forks the process tree including all sub-processes. Then, it assigns the same process identifier (PID) to that process as it had during check-pointing. However, if assigned PID is different, then CRIU is unable to restart the process and restoring process of container is crashed.
- In third step, CRIU restores and prepares some basic resources such as files, sockets, name-spaces and private virtual memory addresses.
- Finally, another piece of code known as "restorer-blob" is introduced to un-map the CRIU code and its memory contents from target process as well as restoring the main-process memory mappings. Similarly, process timers and credentials are also restarted so that CRIU can perform some privilege operations and trigger restore process at a moderate pace.

Figure 2.5 demonstrates the default checkpoint-restore process from source node to destination node. As discussed above, CRIU dumps and writes down the memory of container to disk. Then, a Linux command (scp or rsync) is used to read those files from source and writes them to destination disk. Finally, CRIU-restore reads the copied information to restart a new container on destination. However, this two-time read-write operation starting from checkpoint to restore process is I/O intensive and therefore, can result in increasing downtime when disk I/O is not super-fast. This challenge can be solved by a technique introduced as "disk-less migration" which is discussed in following section.

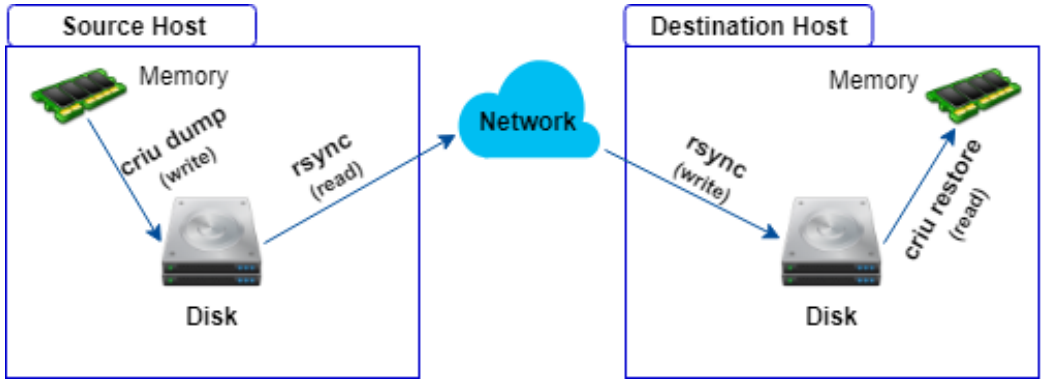

Figure 2.5: CRIU Default Working Principle (adapted from [25])

2.3.2 Live Service Migration Using CRIU

The main application of CRIU is to live migrate containerized applications. In real-time, CRIU can checkpoint and restore containers from a source node to a destination node. For example, it can migrate any type of container (e.g. database server or game server) to remote end until and unless it needs no direct access to low-level hardware resources. In this mechanism, running containers are frozen on source. Their check-pointed image files are copied to destination and finally, they are restored from those frozen images. To achieve container migration, following things are required:

1. Updated and same version of CRIU should be installed on both sides (better to install CRIU version 3.9 or 3.8.1).

2. Docker Enterprise Edition 2.0 must be used and docker experimental feature (i.e. docker checkpoint) must be set to true. At present, some versions of docker CE lacks the desired support to properly checkpoint a container and then restart it from that checkpoint.

3. Container file-system (i.e. binaries, libraries) as well as docker images must be exactly of same version (without any differences) on source and destination. Otherwise, CRIU restore will crash and return the corresponding error.

4. For migrating TCP connections, source and destination should have same IP address [18]. This is due to fact that CRIU enable their bind() and connect() functions based on source credentials.

In case an IP conflict occurs, the corresponding system-call will fail and ultimately the CRIU restore process is aborted.

5. CRIU cannot migrate containers which try to access low level hardware information. If an application requires this access, it can be provided through necessary software plugins on source which help CRIU to extract hardware state and then use the same state on destination using CRIU restore.

As live migration is often linked and imagined with CRIU but in practice they are barely equivalent to each other. This is due to fact that CRIU is just a checkpoint and restore process. It suspends a container and then restart it from check-pointed image files on destination. However, it lacks the ability to move a running container and its memory pages to remote end. Therefore, it involves more delays and downtime when migrating a container. The downtime includes the total time to checkpoint, migrate and restart the container. For complete live migration of Docker containers, an open source project known as P.Haul (i.e process hauler) [22] is proposed. However, at present, this tool is under development and testing stage which is quite slow, so it will take much more time to become a stable and mature tool for live container migration. To envision live migration, CRIU achieves short downtime with the help of following types of migration strategies:

1. **Iterative Live Migration (pre-copy)**
   Live migration essentially requires two things: copying disk file of container and its memory pages. Whenever disk files are copied, container keeps running but moving memory-pages requires the need to stop the container. Thus, if memory pages are of large size (e.g 1 GB), then observed downtime would be critical. To solve this challenge, iterative checkpoint feature was developed. As the name suggests, iterative migration checkpoints the large memory pages in a repetitive manner such that each new checkpoint records only memory changes relative to previous checkpoint. By adopting this approach, multiple repeated checkpoints (known as pre-dumps) are created without stopping the running container except the last one (i.e. dump-phase) which suspends the container′s process. Furthermore, the observed downtime in dumping a container will be decreased by a significant factor as it just involves moving small memory-page to destination.

   Recently, A.Reber [14] integrated this pre-dump feature of CRIU with runc and LXD containers to achieve live migration. How-

ever, Docker still lacks this integration so, at present, only simple dump and checkpoint-restore facility is available for docker containers.

2. **Lazy Migration (post-copy)**
   Unlike iterative migration, lazy migration is a type of post-copy memory migration [25]. In this approach, the memory contents of a container are kept on source node without moving them to destination. Instead of copying everything, only freeze state information and bare minimum is copied over destination to start the application. Later, source node pre-pages [25] the remaining memory sate information to destination. The concept of page-faults is introduced to request the required memory pages from source. Through page-fault handlers, all desired pages are moved to destination while application remains active at the same time. This method also reduces the frozen time of an application. At present, this method is still under development within CRIU community.

3. **Diskless Migration**
   This is another type of technique to reduce the migration time of an application. Instead of dumping a container′s image files and memory to persistent storage, they are placed over temporary file system (tmpfs) which is mounted on both sides [7]. As shown in Figure 2.6, now a CRIU dump will launch a criu-page server on destination side and instead of dumping data to disk, it is dumped to page-server. This page server will receive pages from CRIU and write them to tmpfs mount. In this manner, the data is dumped directly to destination without storing it on source node. Thus, CRIU-restore from this tmps appears to be very fast process due to memory-to-memory copy and eliminating the need to perform two-times read/write and I/O operations.

   Apart from live migration, CRIU can be used for multiple purposes. It can save the current state of a game-server which can be resumed at later stage. Similarly, long jobs are suspended at current state and can be restored whenever desired. It possesses the ability to solve high performance computing issues (HPC) related to cluster load-balancing. This also correlates with one of our objective i.e. to trigger migration and implement load-balancing

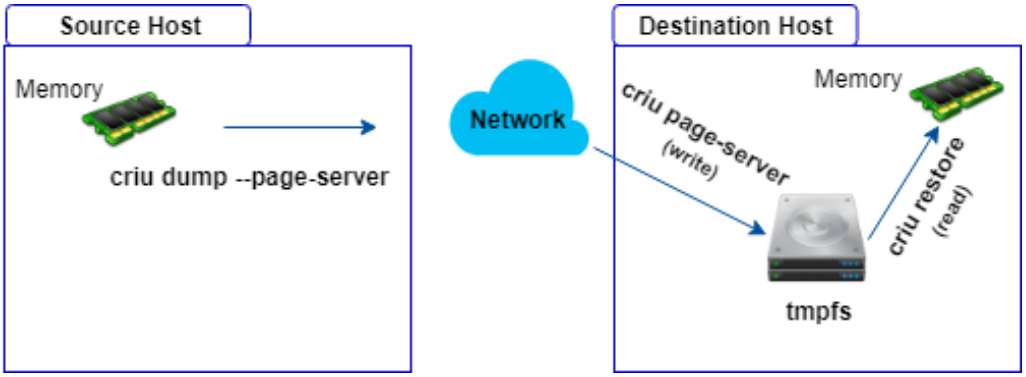

Figure 2.6: Disk-less Migration using Page-Server (adapted from [25])

among Fog nodes whenever they become overloaded. Some other use-cases include building fault-tolerant systems, moving an application into screen and fast kernel upgrade etc.

Chapter 3

Related Work

3.1 Resource Management in Decentralized cloud

Resource Management, mobility challenge and fast service migration (i.e. reduced latency) are some of key challenges in both mobile cloud computing as well as edge computing environment. As cloud virtualization platforms i.e. MCC and MEC both complement each other and there is no perfect solution which can address current challenges by just relying on a single technology. Therefore, in this work, resource management and latency problems are studied from both domains.

In a traditional Mobile Cloud Computing environment, computation-intensive tasks have been offloaded from mobile devices to remote cloud servers to optimize their energy consumption and computation cost. The recent research also demonstrates various techniques and algorithms (e.g. Round robin and first come first serve strategy [44]) to schedule and optimize resource allocation among cloud servers. ThinkAir [36], MAUI [34] and CloudClone [31] execute application code in remote servers and harness their computation capabilities by executing multiple VMs. In addition, energy consumption of tasks before and after offloading are also predicted to reduce the execution time and take on respective cost-effective offloading decisions. Similarly, M. Lagwal and D. Yao in [38, 45] solves this load-balancing issue with the help of genetic algorithm method. At first, VMs and cloudlets are sorted out with respect to their current load and local processing power. A broker then fed this information to a genetic algorithm for allocating necessary tasks to these cloudlets based on their current load and task

handling capabilities. Other methods for load-balancing include traffic engineering such as SDN based programmable Fog Networks. However, these all methods fail to address low-latency (i.e. 1ms) requirement for mobile edge users. Therefore, most of recent research work involves exploring MEC in conjunction with MCC to address this challenge.

Mobile Edge Computing (MEC) provides service in proximity to end-users but on the other hand, it also faces challenge of service migration and resource management among Fog Nodes. The recent research addresses these issues with the help of novel methods and strategies for resource management and fault-tolerant service provision from edge nodes. T. Ojima in [30] presents Kalman Filter method to predict user mobility which can be later utilized for service provisioning from nearest edge node. To solve the problem of edge-node selection, terminals anticipate the user location with respect to task collection time and then fed this task processing request to any nearest edge node. Similarly, [13] designed a resource management algorithm named as MExR (Mobile Edge mixed reality) to enable the execution of mixed reality applications at the edge of network. It can serve many tasks including packet assignment, load balancing and placing computation tasks on network nodes. The major components of this platform include SDN based Ryu controller, NS-3 network simulator and HA-proxy load balancer. At first, workload information of all network nodes (including edge and cloud servers) is sent to a load-balancer. Then, other network parameters such as network latency and current network load are also observed. Finally, a resource allocation algorithm utilizes this collected information to decide on task processing either from edge nodes or from remote cloud servers.

As mentioned in [32], V. Chamola simulated a SDN based network of cloudlets that can offer computation intensive tasks to mobile devices and as a result improves their quality of service. This framework shows superior performance with respect to network latency when compared to earlier existing solutions. In this framework, whenever a fog node exceeds its processing limits, the offloaded tasks assigned by mobile devices are transferred to a nearest fog node. A central cloud manager (sitting on top of SDN network core) is aware of available resources on all fog nodes and thus it enables fast service migration to a nearest feasible fog node. Motivated by this concept, our work also developed a centralized control server, which monitors the resource usage

on each fog node and then dynamically decides about service migration to another available fog node. Similarly, H. Zhang [41] proposed some policies for resource scheduling which allows seamless handover over IPV6 and reduces latency time to some extent. Following the Game theory approach, [46] demonstrates a 3-tier hierarchical model and alleviates resource allocation issue by forming a student project matching game. In this framework, data service operators (DSO) act as leaders and deliver virtual services to authorized data service subscribers (ADSS) through Stackelberg game. A moral hazard model is designed between fog nodes and DSO which help fog nodes to efficiently release physical resources to ADSS.

3.2 Strategies for Container Migration in Edge-Computing

Elijah [43] is one of the famous open-source edge computing platform which provides VM based multi-tenancy environment. It creates cloudlet-specific extensions of Openstack (known as Openstack++) to create and manage cloudlets with VM migration and dynamic VM synthesis. However, this platform adds redundant hypervisor layer abstraction to deliver any software service. As a result, service provisioning and VM hand-off strategy lacks zero-downtime support for latency-critical applications. Therefore, there is need to devise a solution for fast and live service migration based on light-weight and container-based virtualization.

The concept of resource management of fog nodes through container-based live-service migration has gain great importance in recent years. Based on this concept, some projects have been developed which focus on fast service migration among fog nodes to ensure load-balancing and reduced network latency. Some famous container management and orchestration tools (i.e. Docker Swarm, Universal Control plane [24] and Kubernetes [26]) also promise to dynamically initiate and monitor containerized applications on edge devices. These tools enable any physical or VM to join hundreds of other network nodes to form a cluster of nodes which can be monitored and managed from a single interface. However, at present these solutions lack feature of container migration from one edge node to another except Kubernetes which provides pod migration service to edge nodes. As this work focuses on container

migration strategy to manage work-loads among fog nodes, therefore, some of recent container migration techniques are briefly discussed.

A. Machen in his article [39] solves the challenge of migration down-time in live-service migration with the help of a layered framework proposed for LXC containers. The basic method is to divide the application into multiple layers and migrate only those layers which are absent on target node. For this purpose, an incremental file synchronization (i.e. rsync) method is utilized which can easily track missing layer on destination. In its 2-layered model, an application consists of a base layer (i.e. main application package file, OS and Kernel etc.) and an instance layer (running-state of application code). The base layer size is much higher (i.e. almost 90% higher) than instance layer and service downtime greatly depends upon the size of application package. But as same instance of base-layer is needed to start an application, therefore all mobile edge clouds (MEC) possess same version of application package. In this manner, migrating a full-fledged application to any MEC requires no need to transfer the heavy base images. Instead, only instance layer files are transferred which ultimately gives us reduced service downtime due to their small footprint. In an extended 3-layer version of this framework, the instance layer is further subdivided into an intermediate layer which is known as application layer. An application layer encapsulates the application-specific files and data in an idle state. On the other hand, the instance layer now consists of only run-time requirements and in-memory information of an application. Now to migrate a service, at first, the application layer is copied to destination while keeping the service in running state. Afterwards, the service is suspended and instance layer is transferred to destination. Finally, the service is recreated using image layer, an application layer and an instance layer. This 3-layer model migrates most of service′s data before freezing it, and hence achieves low service down-time because live migration involves transfer of only run-time state information.

As proposed in [40], Voyager presents a just-in-time (jit) live service migration based on CRIU′s page-server model and is marked with zero down-time data federation capabilities of union filesystems. It migrates a container even before transferring its whole filesystem. Compared to default CRIU-dump process, Voyager is based on tmpfs (temporary filesystem) and avoids two times disk transactions from source to destination by directly dumping the container filesystem to destination. Before checkpointing a container, the data federation strategy is

imposed with remote-reads on destination without even transferring the actual data to destination. With this data federated orchestration, the rootfs (root filesystem) of a container is exported to destination using union mounts. Later, whenever container is resumed at destination, it performs lazy replication mechanism to synchronize all available files on source with destination and copy only those files which are missing on destination. This data replication process contributes zero downtime to overall service downtime so that Voyager′s total downtime includes the time to checkpoint and restore the container.

Similarly, Cloud4IoT [33] represents another edge computing IoT platform to perform horizontal and vertical Docker container migration in a Kubernetes cluster. This platform is a 3-tier model consisting of cloud, edge and IoT gateways. The end-users discover nearest gateways and establish connection based on BLE (Bluetooth low-energy) communication. Whenever a user is discovered, an event is sent from a gateway to cloud orchestrator (a central management entity) which then starts the service provisioning container on corresponding gateway. The concept of horizontal migration is introduced to provide roaming support to users who leave one IoT gateway and enter the coverage area of another IoT gateway. In this process, their movement is observed with respect to their RSSI (received signal strength indicator). Incase of weak-RSSI, a user-leaving message is generated at one gateway while user-entering message is received on another gateway. Both gateways forward the corresponding messages to central orchestrator, which then changes the node affinity of corresponding container i.e. it stops the container on previous gateway and starts a new container on newly discovered gateway (i.e. stateless migration). On the other hand, vertical migration is introduced to migrate services from IoT gateways to edge or cloud in case the IoT gateways exceed their computation limits. As gateway devices are computation limited devices, so they can accommodate only a couple of end-users, therefore they need edge or cloud migration support to deliver services to all end-users. Another platform [39] also promises to deliver container migration for Docker containers. Based on a MQTT Broker, an IoT device can communicate with edge and cloud. To offload any container from one IoT device in case of overload, the cloud orchestrator finds out the available node by publishing its IP address to MQTT broker. Then docker checkpoint and restore process is executed to restart the container on destination node. This work also shows an exponential increase in migration time when container image size is increased because large images demand more

time to download them from DockerHub.

As stated in [42], CRIU has been chosen as a feasible migration tool for LXC containers due to its checkpoint and restore capabilities. However, this work replaces traditional TCP protocol with multipath TCP (MTCP) and integrates it with CRIU working to achieve fast migration with reduced service down-time. Unlike traditional setup, destination node is assigned two interfaces to establish connection with the source. Now whenever, a running container is check-pointed, and its memory state and file-system are transferred to destination, this process get completed very fast because now we have multiple sub-flows to copy container contents from source to destination. However, this work stated that LXC containers find no support for liver container migration because the migration tool (i.e. CRIU) lacks the ability to live migrate an application as its basic feature is to suspend a running application, then save its state information and copy all contents to destination. Only then it can restart that application from same state on which it was check-pointed. This is due to fact that it lacks feature of iteratively dumping an application (i.e. iterative pre-dump feature which is realized in upcoming container migration tools (i.e. Flocker [10] and P.Haul [22]).

Recently, Adrian added pre-copy support for runc and LXC/LXD containers [21]. It issues lxc-move command to live migrate an application to the specified destination. This single move command execution involves many steps. At first, the container file system is synchronized on source node and pre-copy checkpoints are created multiple times without suspending the container. For default configuration of LXD, the final checkpoint is performed only in case when either the number of pre-copy iterations reach 10 or they account for at least 70% of memory pages of container. This final checkpoint instance contains relatively small memory changes compared to earlier pre-copy checkpoints. After migrating final checkpointed image files to destination, the container file-system is again synchronized to ensure that same version of container file-system exists on destination. Finally, CRIU can restart container from those checkpointed files within zero-downtime. In some cases, downtime may also increase if final memory page size is large.

Motivated by pre-dump feature in LXC based containers, MIRA framework [29] developed by A.Ramy and Dutra also considered CRIU for fast service migration in 5G networks. It discussed and compared

the stateful migration scenarios from source to destination with respect to pre-defined and undefined paths of migration. The pre-defined path of migration involves two famous migration techniques (i.e. tmpfs and disk-less migration) which have been integrated with iterative dumping process. In tmpfs migration, the memory information of container is iteratively dumped and then copied from source to destination whenever final checkpoint is performed. On the other hand, disk-less migration solves the issue of downtime by starting a page-server on destination and performing remote-reads repeatedly. Even though these methods effectively reduce the migration time but still less efficient when compared with undefined path migration. This is because copying of file-system and memory information is performed during migration phase which results in longer downtime. In contrast, when there is no specific path defined from source to destination, then all MECs are assigned a shared network file system (NFS). In case a container needs to be migrated from one host to another, then it only needs to copy memory state information while container file-system is already accessible through shared storage. This method greatly reduces the downtime and interruption in live service migration (as observed in this work) but it achieves this at the cost of increasing overall network utilization.

Table 3.1 demonstrates the relative comparison of each of the service migration platform with respect to its service-migration strategy, support for zero downtime migration and virtualization environment. It shows that traditional VM based platforms such as Elijah have no built-in support for low-latency service migration. On the other hand, famous container-based solutions also take more than 2-3 seconds to migrate the service. At present, only MIRA project claims to ensure service migration near zero-downtime (i.e. under 1 second). From container-environment perspective, all these projects achieve service migration using low-level containers (i.e. runc, lxc/lxd) except Cloud4IoT, which utilizes high-level containers (i.e. Docker containers) for service migration. As our project demands support for Docker based applications therefore, current work focuses on service migration using Docker containers. For this purpose, docker′s experimental feature docker-checkpoint is used in conjunction with CRIU. At present, it only creates one-time dump of a container and lacks the support for CRIU′s optimization tools. Therefore, our first target is to make a proof-of-concept using CRIU′s default pre-dump feature and then experiment the emerging optimization strategies.

Table 3.1: Comparison of Container Orchestration Platforms

Platform Name	Migration Strategy	Low-latency Support	Virtualization Environment
Elijah [43]	VM Migration, Dynamic VM synthesis	No	VM
Kubernetes [26]	pod migration	No no proof of concept	Docker
Layered Framework [39]	incremental file-sync. (rsync)	No latency >=2 sec	LXC,KVM
Voyager [40]	just in time migration (page-server)	No latency >=2-3 sec	runc
Cloud4IoT [33]	Hozrizontal and Vertical migration	No Findings	Openstack, Docker, Kubernetes
MPTCP for lxc [42]	CRIU and multipath TCP	No latency = 242 sec	LXC
Adrian Reber [21]	pre-copy (lxc-move)	No latency >= 2-3 sec	LXC/LXD
MIRA [29]	tmpfs, Diskless Migration, Iterative pre-dump	Yes latency >= 1.042 sec	LXC

Chapter 4

System Design & Implementation

After summarizing the related research work in previous section, now this chapter focuses on describing the overall system architecture, protocol design and mechanism of autonomous migration with respect to two algorithms.

4.1 System Architecture

Figure 4.1 demonstrates the overall system architecture of the testbed developed for resource management. This testbed comprises of a public cloud , three edge nodes and a central nodejs server to control all nodes. For cloud-layer emulation, a Microsoft Azure VM (named as mec) is deployed (from Europe region) with required specifications. Three physical machines (LAB3, LAB4 and VRONE) from Aalto campus research network (research.netlab.hut.fi) form the edge cloud layer which is further subdivided into two different categories (i.e. Edge Cloud A and Edge Cloud B) with respect to their network information. Then, a nodejs server runs on top of a physical machine which lie next to edge nodes. The system specifications of all edge nodes and cloud device are clearly mentioned in the system design. To reach each edge node and remote cloud with minimum delay, the control server establishes a persistent-SSH connection with them. In addition, cloud and edge nodes can also access each other through SSH-configured connections. In this setup, nodejs server acts as the master node and all other edge devices and cloud act as clients. The real-time event-based information

exchange occurs between server and clients through socket.io server-client model which is discussed in following section.

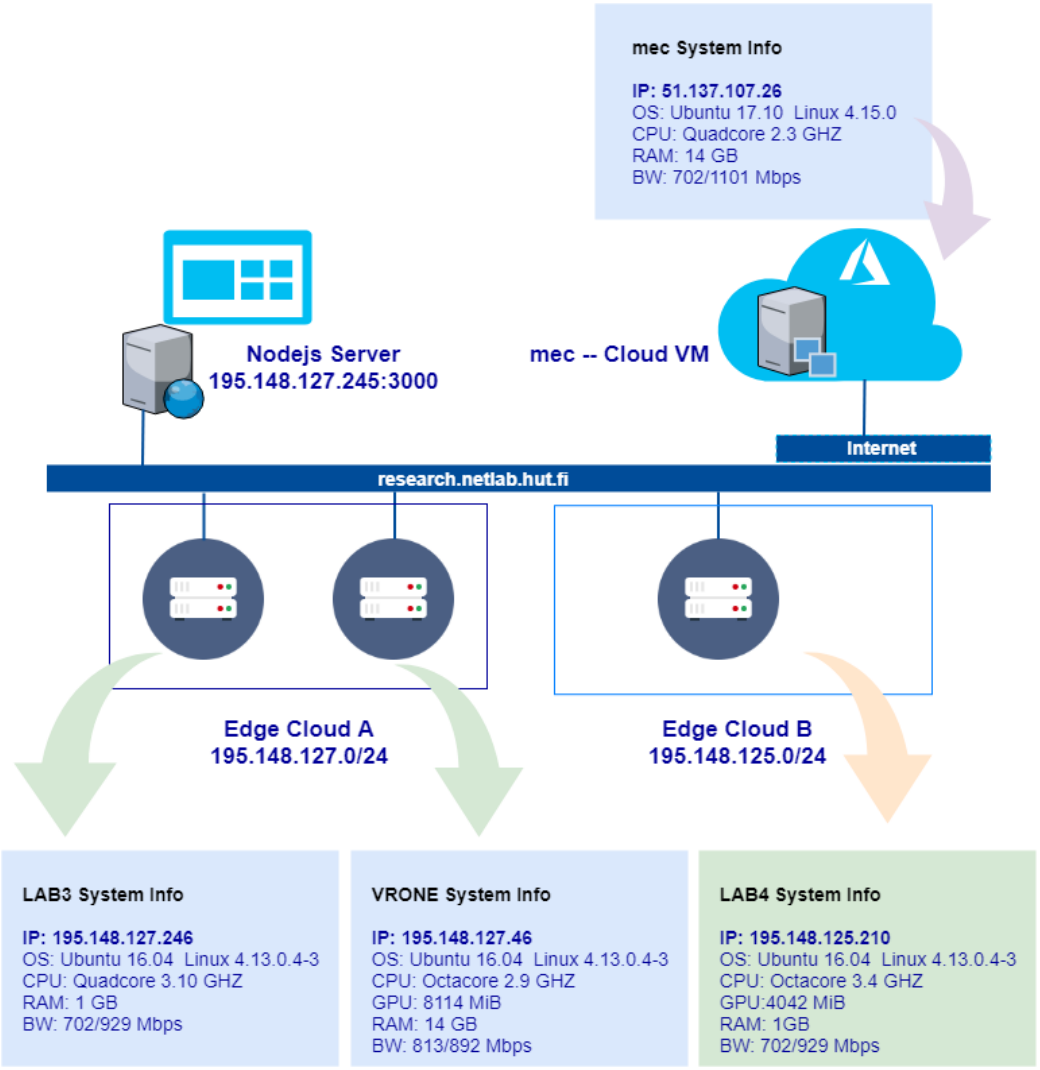

Figure 4.1: System Architecture of container-based MEC Platform

4.2 System Dashboard

Figure 4.2 shows an interactive and user-friendly application dashboard which is developed to monitor the real-time state of all nodes and dynamically perform container orchestration service. Inside main window, it shows the list of registered nodes, application profiles and

all deployed applications. Then, a main menu on top-left gives user the freedom to perform various operations such as node registration, application profile handling, container service initiation and migration. Before going into detailed description of dashboard features and its

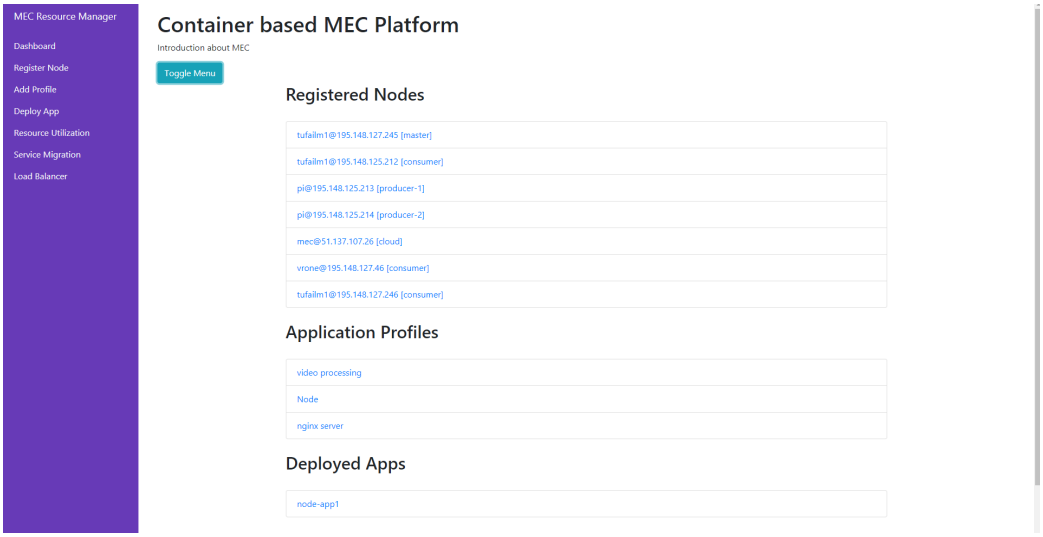

Figure 4.2: Application Dashboard

work-flow, it is important here to first introduce the basic platform and tools on which this whole dashboard is developed.

4.2.1 Front-end Technologies

The front-end of dashboard comprises of traditional web technologies such as HTML5 (hypertext markup language), CSS (cascading style sheet), Bootstrap and JavaScript (JS). HTML5 is used to construct the structure of each webpage of dashboard. Along with it, CSS is used to impart styling features such as font-color, size, text-formatting etc. Then, JavaScript language comes to put action into webpages and make them interactive and responsive. This is a multi-paradigm language which is used for both client and server-side implementation. Another important framework used is bootstrap, which is an open-source library to create responsive web and mobile-first applications using pre-built styling classes.

4.2.2 Back-end Technologies

On backend, it uses node execution run-time environment to build the web-server (i.e. nodejs server). Based on asynchronous JavaScript, this platform follows a single threaded, event-driven and non-blocking I/O model to give superior performance (in terms of speed and system resources) for real-time data intensive web applications. This also corresponds to our requirement of lowest delays while performing CRUD (create, read, update and delete) operations. To persist the application data, a local database is created using mongodb driver engine. As dashboard application requires RESTFUl API to create, store and fetch data from a local database, so for this purpose, express framework is used. On top of nodejs, this framework provides declarative routing (i.e. routes) to perform HTTP requests (GET, POST, PUT, DELETE) on specified URL identifiers. Furthermore, this application follows nodejs MVC (model, view, controller) architecture to process HTTP requests, consult with database and then render the response to user.

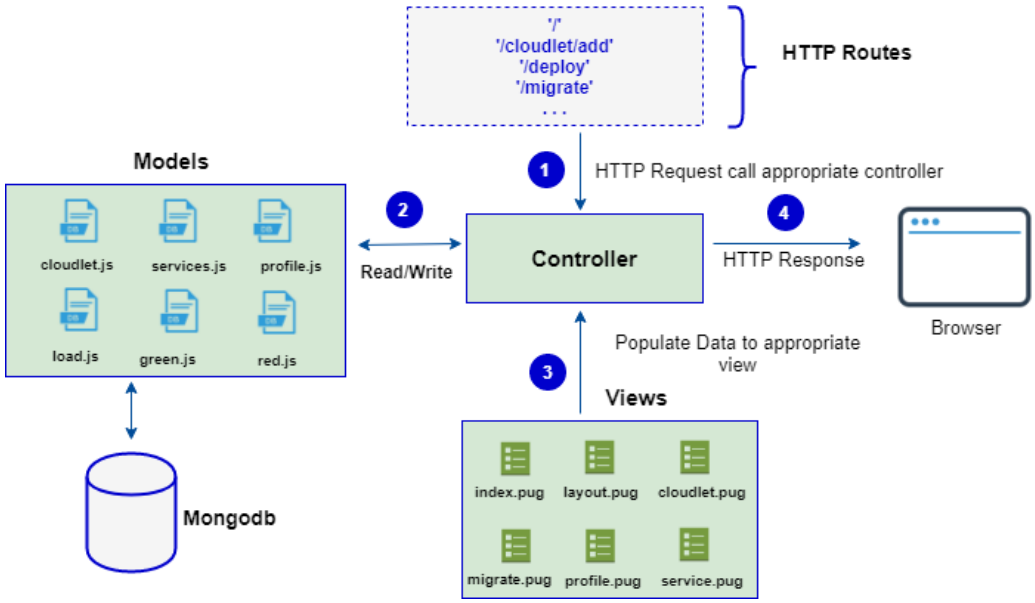

Figure 4.3: Application MVC Architecture (adapted from [9])

Based on MVC-architecture as shown in Figure 4.3, this application consists of three main components (i.e. Model, View and Controller) which are listed as follows:

- **Models:** define the proper schematic layout of application data which needs to be stored in database entries. They are also re-

sponsible to connect with database (through an object relational mapper) and transform its contents as valid re-presentable objects. As demonstrated in Figure 4.3, our application employs six models to read and write information related to registered nodes, deployed services and application profiles.

- **Views:** They are re-structured templates which get data from models (through controller) and then render it to user. In current setup, template engine pug is used to acquire data from models and then generate the user interface.
- **Controller:** It enables the interaction between views and models. Based on HTTP requests, it triggers appropriate control functions to fetch data from models and then show it with corresponding views.

4.2.3 SocketIO for Real-time Communication

Socket.IO [27] is a library used to establish real-time and bi-directional communication between client and server. It emits and listens on certain events (also known as socket.io events) to exchange real-time data. This tool is chosen in current setup because it establishes reliable connection even in the presence of proxies, system firewalls and load-balancers. Furthermore, in case of disconnection, a heartbeat-mechanism informs the server and client that a connection failure has occurred. Then, the client will keep trying to reconnect to server based on automatic reconnect support.

In current setup, Socket.io consists of a nodejs server (i.e socket.io) and a JS library (i.e. socket.io-client) for client-side implementation. An object **io** is defined as a new instance of socket.io server and socket.io-client. On server side, the express framework is used to start the server on http port 3000. Then this server instance is passed to socket.io engine so that it can listen and respond to connection requests from socket.io clients. An **io.connect()** method on client-side (i.e. on all edge nodes and cloud device) takes the server address and port as input parameters to connect to socket.io server. Finally, the server listens to that socket.io connection with **io.on()** method and establishes a real-time socket-connection with each client. To perform real-time event-based information exchange, **socket.emit()** method transmits an event with certain information and the other side listens to the same

event with **socket.on()** method to retrieve the transmitted data. Depending upon our requirements, the current setup contains following socket.io events:

- **set.role@cloudlet[i].ip**
  Server defines this event to assign designated role to each registered node. Here, cloudlet matrix contains information about all system nodes so socket event

  ```
  socket.emit('set.role@cloudlets[i].ip',{role:cloudlets[i].role})
  ```

  emits role information of every node . Then, each node retrieves this information with event-listener

  ```
  socket.on('set.role@nodeIP', data=> {})
  ```

  Here, the nodeIP is the IP address of an individual node and data object contains the information about assigned role from the server.

- **send.load**
  When all nodes have successfully received their designated roles, then nodes with role as edge-node will send their real-time system utilization information (containing CPU, GPU and RAM usage) after every 5 seconds to server, which stores and update this information at real-time.

- **start.req@destination.ip**
  This event is emitted from server to start a container on the specific node with given IP address. The destination node listens to this event using

  ```
  socket.on(start.req@nodeIP, data=>{})
  ```

  method, retrieves the transmitted parameters (including container name, image, arguments) inside "data" object and then a bash script deploys the container from those parameters.

- **stop.req@destination.ip**
  This event is triggered to stop a running container on any specific node. It executes the stop-script and suspends the container service.

- **start.res/stop.res**
  When a container is deployed on any node, then start.res() event is emitted to sends back the server a response which contains information whether container was successfully deployed or it encountered some error. Similarly, stop.res() sends back the server execution result of container-stop script.

- **mig.req@source.ip**
  This event is defined to migrate a container from the source node with specified IP address. It contains information about container name, image, arguments and IP address of destination node.

- **mig.res**
  This event transmits the container-migration response to server and informs whether migration was performed properly, or it involved any issue.

4.2.4 Container Advisor (cadvisor)

Container advisor (cadvisor) [3] is a container-based tool developed to collect and export real-time resource usage of a machine (physical or virtual machine). Kubernetes [26], a famous container orchestration tool, also utilize cadvisor to monitor the system performance and resource utilization. With its native support for Docker containers, cadvisor can show the list of active containers on the system and also provides system utilization information for each container. In current setup, cadvisor is deployed as a docker container (on all active nodes) which starts a web-server accessible on HTTP port 8080. Then, an iframe [15] window is used to embed this web-server inside the main view page of each registered node. Therefore, whenever the user clicks on any registered node (e.g. lab3 in shown figure), a new view is rendered to display the node-level information along with system-usage statistics collected by cadvisor. Figure 4.4 shows cadvisor capabilities to show running Docker containers, active processes, system usage information (i.e cpu, memory usage and storage information) and network information. If the "Docker containers" button is clicked, then it gives the possibility to view occupied system resources by each container.

Figure 4.4: System Monitoring with cadvisor

4.2.5 Docker Checkpoint for container Migration

Docker checkpoint is an experimental feature of Docker, which is used in conjunction with CRIU to freeze a running container and then resume it on the destination host from same state at which it was checkpointed. This whole process of container checkpoint and restore is done to migrate a container from one host to another. For this purpose, our migration script adopts Shaun [6] strategy and achieve container migration based on following steps:

1. **Checkpoint**
   At first, migration script is provided with essential information about the active container (such as its name, image, arguments and destination host). Then, to create a checkpoint for container e.g checkpoint_A for containerA,

   ```
   docker checkpoint create ${containerA} ${checkpoint_A}
   ```

   command is used. This command stops the containerA and saves its memory information in the form of .img files at container location /var/lib/docker/containers/containerA_ID/.

2. **File-Transfer (using rsync)**
   The second step starts with copying container-checkpoint files from container directory /var/lib/docker/containers/containerA_ID/checkpointA/ to /tmp directory. For this purpose, a tar package is used which compresses the checkpointed folder (i.e. checkpointA) and moves it to /tmp directory. Then rsync, a fast file-transfer tool is used to transfer the checkpointed folder from /tmp directory to destination/tmp directory.

3. **Restore**
   When the check-pointed compressed file is successfully transferred to destination host under /tmp directory, then using SSH connection, a new container is created on destination using source container credentials. The command,

   ```
   docker create --name containerA ${arguments} ${container_image}
   ```

   is used to create a new container on destination host. Then, the check-pointed contents from /tmp directory are uncompressed and transferred to new container checkpoint directory /var/lib/docker/-containerA_ID/. Finally, the command

   ```
   docker start --checkpoint containerA checkpoint\_A
   ```

   will start containerA from same check-pointed state at which earlier container (i.e containerA on source host) was frozen. Even though, the container migration can be marked as complete after this step but, the migration script continues and removes the

check-pointed files from /tmp directories of both source and destination hosts. Then, the earlier created checkpoints are also removed. This is done to avoid checkpoint-name conflict which may occur in future container migration, if the checkpoint with same name already exists on any host.

4.3 System Work-flow

The application dashboard is designed to perform following core functions:

4.3.1 Edge-Nodes Registration

Figure 4.5 clearly demonstrates the complete work-flow to register a node.At first, the user clicks the "Register Node" button, which loads the controller function **app.get('/cloudlet/add')** to render the corresponding view (i.e. cloudlet.pug). This view contains HTML form to input the required fields (i.e. name, ip, ssh-key, role) for node registration. The drop-down option role offers three types of role (i.e. edge, cloud and client) to system nodes. After entering the node-related information, a **Submit** button posts this information using controller function **app.post('/cloudlet/add')** and saves it into database in the correct format defined by model schema coudlet.js.

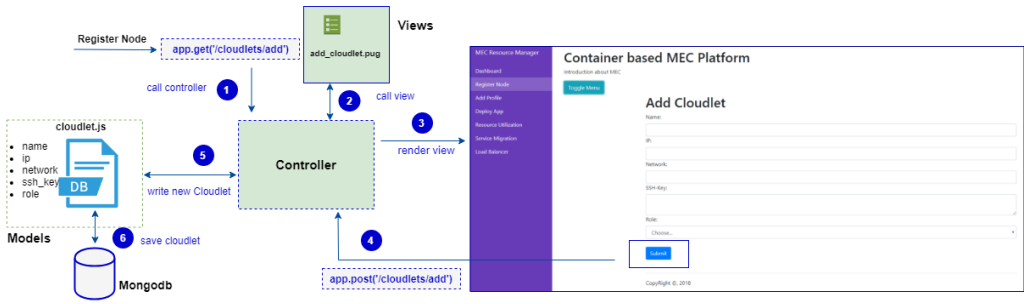

Figure 4.5: Edge Node Registration Work-Flow

4.3.2 Application Profiling

Application profiling works like edge-node registration feature, except that in this case, the user submits a form containing application profiling information which is then saved into database using model schema

profile.js. This feature is developed to assign a limited quota of system resources (i.e. CPU usage, GPU, RAM) for each containerized application with respect to its base image. This is also utilized in autonomous container migration to ensure that destination host possesses enough system resources required for smooth service operation.

4.3.3 System Monitoring

As discussed earlier, cadvisor is the tool used to show the real-time system utilization in the form of dynamic CPU and memory meters, graphs and histograms. However, all edge-nodes possess separate modules and python scripts which measure real-time system usage at every 5 seconds and then fed this information to nodejs server. The server utilizes this information to filter-out edge-nodes as normal or over-loaded. The edge-nodes with system-utilization greater than a certain threshold (i.e 50%) are marked as over-loaded or Red nodes whereas, the edge-nodes with system utilization less than this threshold are marked as stable or Green nodes.

4.3.4 Container Deployment

To deploy a container on a registered node, the "Deploy" button on main-menu triggers the controller function **app.get('/deploy')** and renders the corresponding view (i.e. service.pug), which takes user input including target node address, application name, type, image and arguments. In the input form, the user can select service type as stateful or stateless. The type **"stateful"** means that whenever this container is migrated to another host, then it requires transfer of all memory contents to destination side. However, the type **"stateless"** means that container will be suspended on source host and then restarted on destination from its initial state. This type is useful for scenarios where containers perform real-time operations without requiring any earlier memory contents.

Then, as shown in Figure 4.6, the **"Deploy"** submit-button posts this information to the server using controller function **app.post('/deploy')**. Inside this function, an event **"start.req@nodeIP"** is emitted towards the corresponding edge node to start a container with input parameters. The edge node, then listens to this event and forwards the container-start parameters to a bash script, which ultimately starts the container using **docker run** command. Another event, **"start.res"** tracks the execution result of script and sends back the server a response to inform

whether the container was successfully deployed, or it incurred some error. In case of positive response, the server follows service.js model to store container-related information in the correct format (i.e. node name, application name, image, type and arguments). On main menu, **"Deployed Apps"** list also utilizes mongodb service collection to show all newly deployed containers of the system.

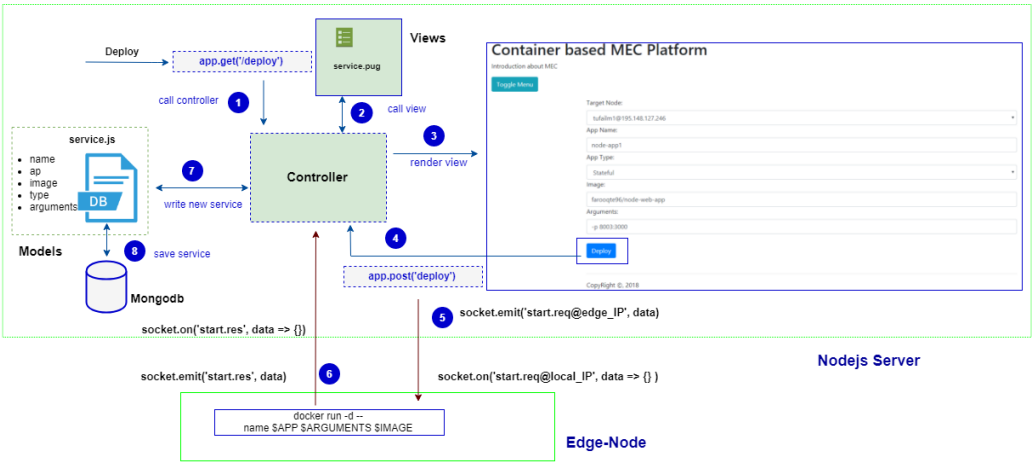

Figure 4.6: Container Deployment Process

4.3.5 Container Migration

From dashboard, container migration process follows the same workflow as the container-deployment phase. However, now the user only selects some parameters (such as deployed application name, source node and target node) from drop-down options. Then, the "Migrate" button submits this information using **"app.post('/migrate)"** function. As depicted in Figure 4.7, this function consults with service database to find out the application type and then take the following actions:

1. If the service-type is state-less, then container is stopped on source host and re-created on destination host. For this purpose, "stop.req@source_IP" event is emitted towards source and "start.req@destination_IP" is emitted towards destination. These two events get the required information from database and then return corresponding responses to server i.e. "start.res" and "stop.res".

2. If the deployed application is stateful, then server emits the migrate event "mig.req@source_IP" towards source host and com-

pletes the migration process in three steps (i.e. checkpoint, file-transfer and restore) which are already discussed in earlier section about docker-checkpoint. After container-restart on destination, a migration response event "mig.res" is sent to server to inform about migration result.

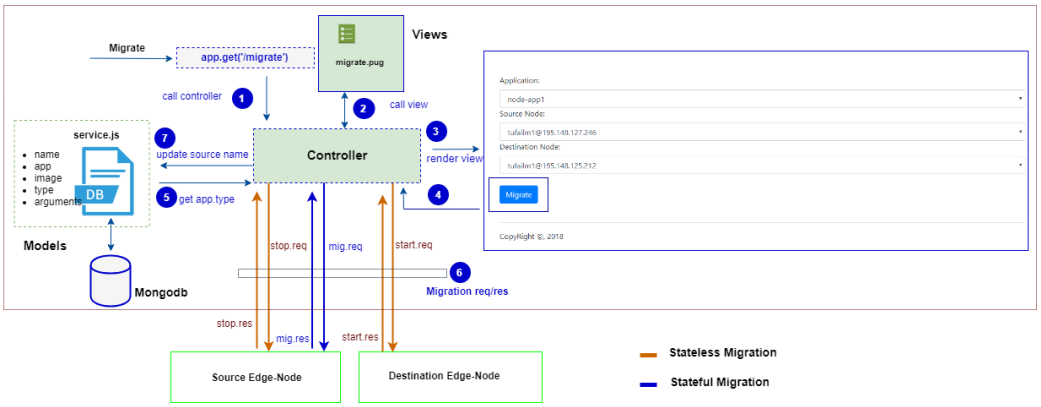

Figure 4.7: Container Migration Process

4.4 Communication Protocol Design

In previous section, this chapter briefly discussed the system architecture and constituent technologies to build the system. Now, this section will present the protocol design of system and explains how server communicates with client-nodes to perform various operations such as system monitoring, container deployment and autonomous service migration.

Figure 4.8 demonstrates the protocol design of system consisting of server and all clients. In this setup, client-nodes are already registered with server using **"Node-Register"** feature described earlier. Except mec, a cloud device, all other system nodes (i.e. lab3, lab4 and vrone) are registered as edge-nodes. Now, when the server and all client nodes start their operation, they follow a sequence of steps which are briefly described as follows:

1. **Connection Establishment**
   As a first step, server starts listening on HTTP port 3000 and all client nodes will be able to access socket.io server using **io.connect()**

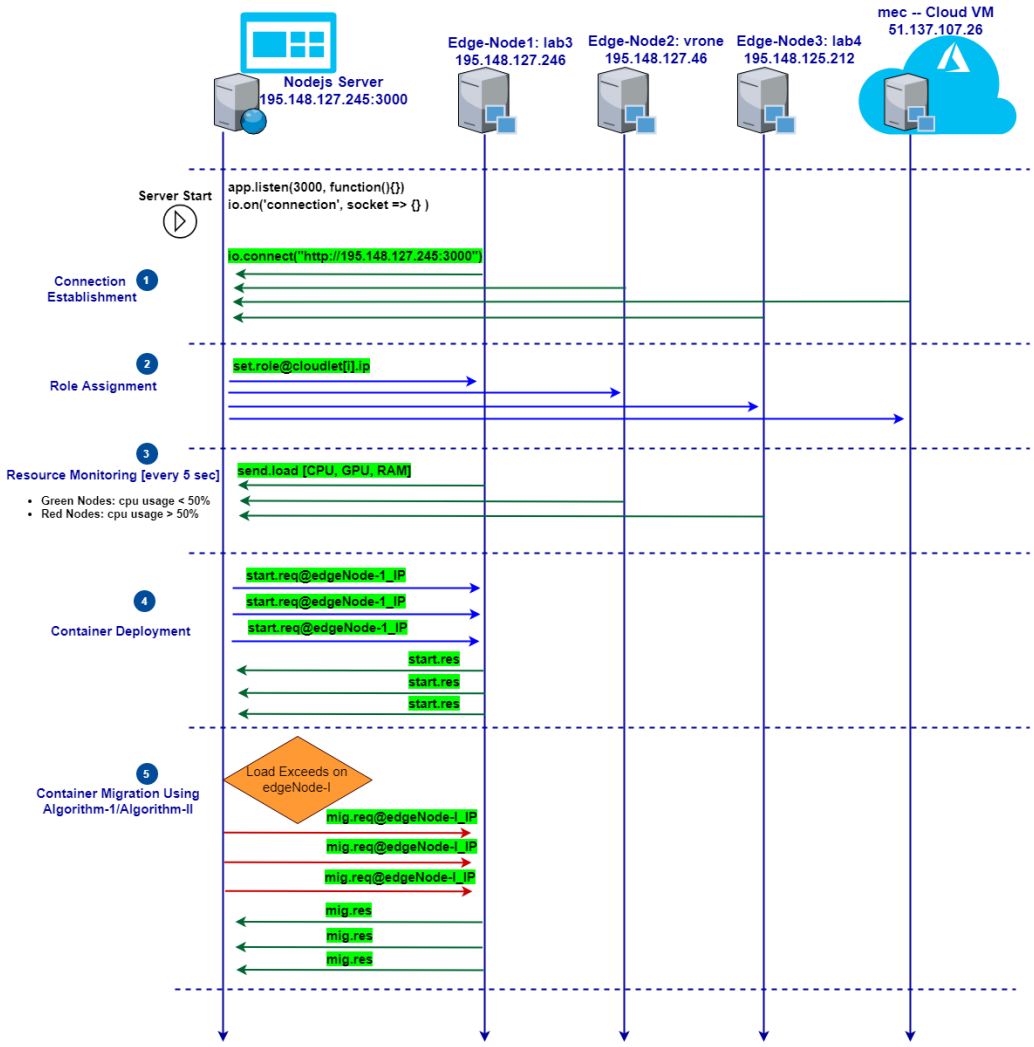

Figure 4.8: Protocol Design Of System

method. When server successfully establishes the socket.io connection with client, it prints out a message to user that a socket connection is opened for a certain client.

2. **Role Assignment**
   As soon as client nodes establish a connection with server, they start listening for event **set.role@node_IP** to perform designated role defined by server.

3. **Resource Monitoring**
   As in current system, edge-nodes are responsible for service provisioning, so their real-time state information must be monitored to take timely action in case of system failure. For this purpose, only clients acting as edge-nodes would transmit their real-time system utilization information (i.e. CPU, GPU and RAM percentage) to server after every 5 seconds. Then, server save those values in load schema and depending upon CPU utilization percentage, it filters out nodes as green nodes or red nodes. Those nodes with CPU utilization percentage greater than 50% are labeled as green while, nodes for which CPU utilization exceeds 50% are marked as red nodes. A rescue function is also defined here, which keeps searching for red nodes and is responsible to take an appropriate action if some node becomes red.

4. **Container Deployment**
   Now, user can deploy some containers on any desired node. (e.g. on lab3). As expressed before, **start.req@node_IP** event will take user inputs and start the container on target node. Then **start.res** event will send back corresponding response to server for each container.

5. **Container Migration**
   As server is continuously monitoring all edge-nodes, so as soon as any edge-node CPU utilization exceeds 50%, its record is removed from green node and added to red one. Then, the server searches all deployed services with the same node address as the red-node. If found, then it gets service deployment parameters (i.e. appname, image. type, arguments) for each service and follows a couple of resource allocation algorithms to decide the most feasible green node. Finally, **mig.req@destination_IP** event migrates each service from affected node (i.e. the red one) to selected green node. For each service-migration request, a corresponding migration response event (i.e. mig.res) informs the server whether migration is performed successfully, or it involves some errors. To understand how destination nodes are selected, the proposed resource allocation algorithms are briefly discussed as follows:

   - **Resource Allocation Algorithm-I**
     For service migration, this algorithm gives priority to nearest edge nodes from same network and finds a green node with least system utilization. If this node lacks sufficient

system resources for service provision, then a least utilized green node is selected from other available networks. The current setup designates that node as dgreen. In case, this node is also fully utilized or unavailable, then services are directly migrated to cloud device (i.e. mec). The flow-diagram in Figure 4.9 clearly shows that algorithm-I performs node-selection mechanism in an iterative manner for each service of red node. At first, it looks for available green nodes from same network. If they are unavailable, then second priority is given to available edge node from different network. In case, they are also busy, then as a last option, the cloud device is selected as the destination node for all services.

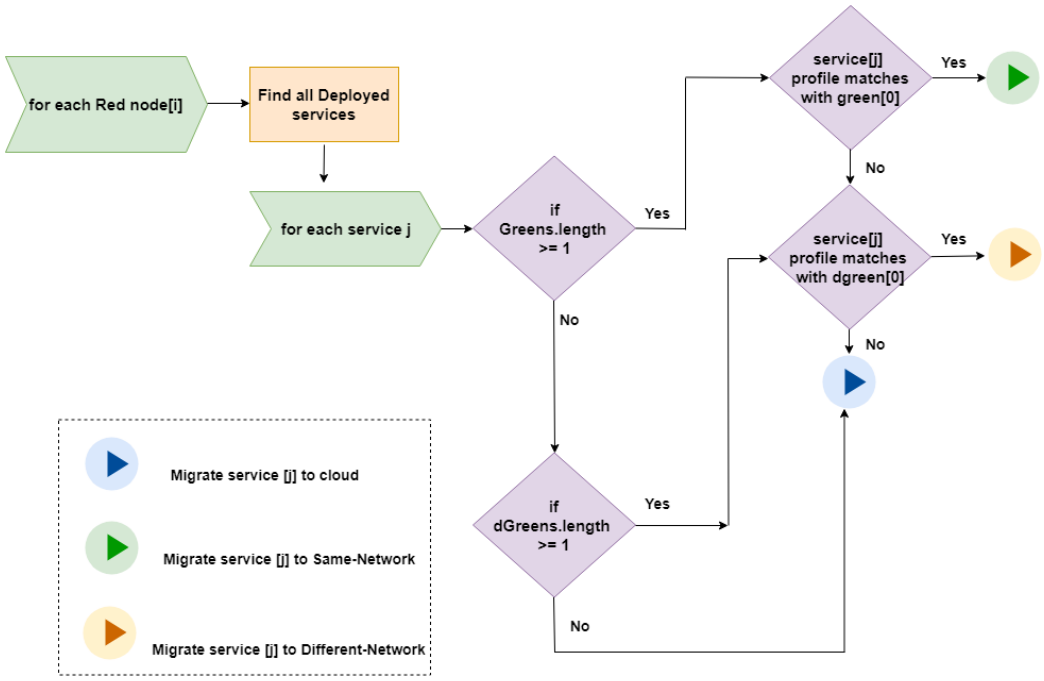

Figure 4.9: Resource Allocation Algorithm-I

In case, if green node from same network is available at the first instant, then first application′s required CPU usage (defined in application profile) is added to green node′s current CPU usage. If their aggregated value becomes less than 50, then it means that application profile has matched with current green node and hence, this service can be migrated to it without causing congestion. Similarly, the remaining services follow the same process of node-selection and algorithm

will migrate them to same green node until and unless their application profile matches with it. Otherwise, dgreen node is checked and all those services are migrated to it whose application profile matches with it. If no dgreen node is available or application profile involves mismatch, then all remaining services are migrated to cloud.

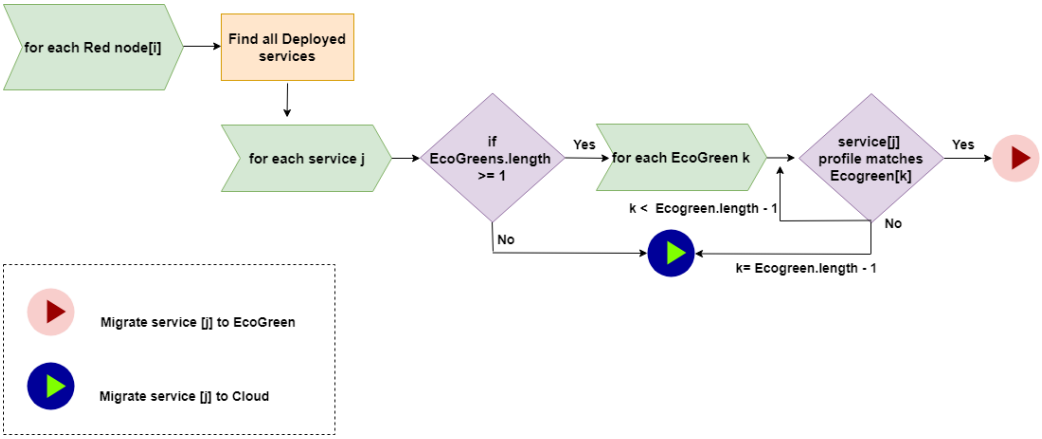

Figure 4.10: Resource Allocation Algorithm-II

- **Resource Allocation Algorithm-II**
  This algorithm aims to achieve green cloudlets to save energy consumption of the system. The idea is to sort out all available edge nodes in descending order of their cpu utilization and select those nodes first as destination host whose system utilization is relatively higher. By doing this, some edge nodes can be switched to sleep-mode or turned off to save energy consumption. For this purpose, each service loop through each node to match its application profile. As shown in Figure 4.10 , a service migration is performed if application profile is matched with corresponding edge node (also known as Ecogreen node). In case of mismatch, the service will look for next available edge node to match its application profile until and unless the last available edge node is reached. If service profile matches with last edge node, it is selected as destination node and corresponding service is migrated. However, if last available edge node also involves application profile mismatch, then service is migrated to cloud. At present, this algorithm works without moving any under-

utilized edge node to sleep mode. However, in future, some power-saving settings can be defined which switches any under-utilized edge node from active state to sleep mode and move its services to a near-by edge node. Depending upon the need to deploy new services or support the existing ones, an edge-node can be later switched back to active state and start delivering the service to end-users.

Chapter 5

Performance Evaluation

In previous chapter, we discussed the system architecture, protocol design and application work-flow. Now, this chapter focuses on validating the design of developed system and comparing its efficiency with respect to service initiation and migration time. To evaluate the overall system performance and understand its behavior, various experiments have been performed and obtained results are discussed in the following sections:

5.1 Deploying & Migrating Stateful Applications

Before deploying any container service, we first add its corresponding application profile information to associate the desired system usage quota in terms of CPU, GPU and RAM percentage. For a nodejs counter application, we register a corresponding application profile with name "node-counter"and 10% usage quota each for CPU, GPU and RAM. Then, we deployed a container on lab3 with name "node-app1" as shown in Figure 5.1. Once deployed, this containerized node application starts on port 3000 and then it is exposed on lab3 HTTP port 8003. Figure 5.1 shows that we can access this counter application on lab3 and for each new page-load request, it increments the counter by 2. Now, using "Migrate" option, this container is suspended at state where the counter is 15 and then restarted from same state on lab4. It took 3.2 seconds to migrate the container and when accessed from lab4 it started from 17 instead of 0. This validates that stateful migration works properly and keeps the memory state while moving a container.

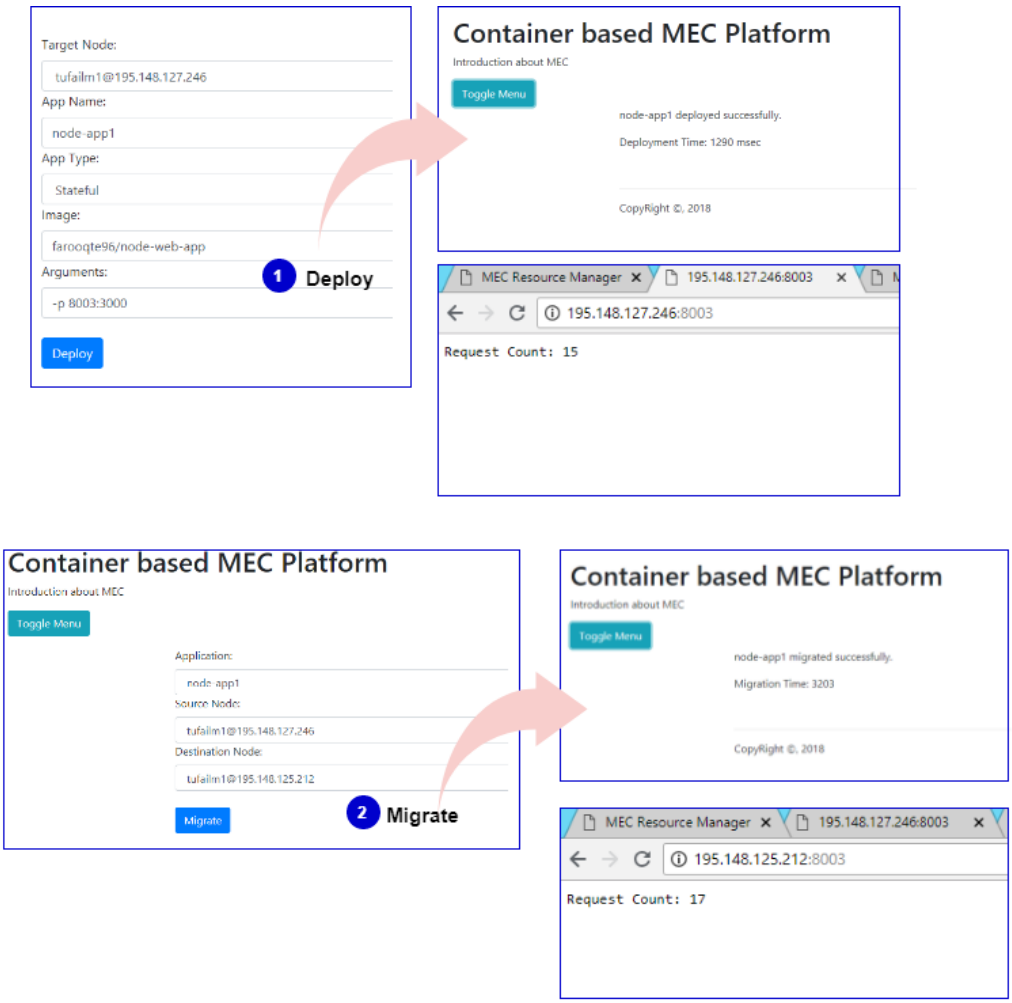

Figure 5.1: Deploying and Migrating a nodejs Counter Application

Similarly, the stateful migration is performed for famous 2048 puzzle game. As this involves migration of an active TCP connection, so IP address should be same on source and destination hosts. However, with default docker bridge, new containers can be assigned only an IP address from default IP pool and it lacks flexibility to assign a user-specified IP address for a container. Therefore, we first created a new customized bridge network on source and destination with name "mec" and subnet 172.18.0.0/16 using command

```
docker network create -d bridge
--subnet=172.18.0.0/16 --gateway=172.18.0.1 mec
```

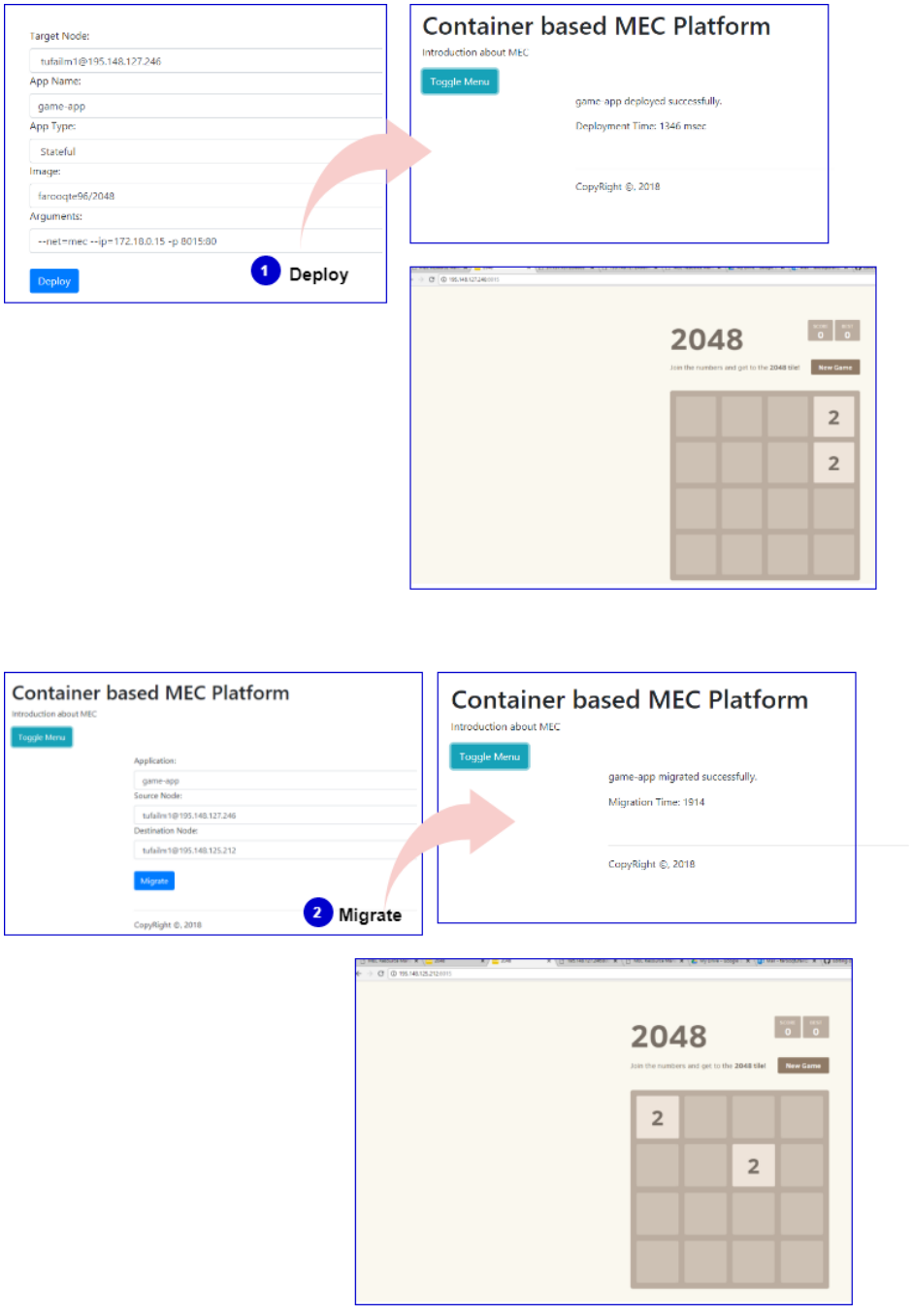

Figure 5.2: Deploying and Migrating 2048 Game

As demonstrated in Figure 5.2, we started a container "game-app" on lab3 with specified IP address and then successfully migrated it towards lab4 within 1.9 seconds.

5.2 Stateful Vs Stateless Migration

To study the impact of service type on migration downtime, both stateful and stateless containers are deployed and then migrated from each source node to each other available destination host. For this experiment, nginx servers are deployed using official nginx base image with 109 MB size.

5.2.1 Stateful Migration

In case of stateful migration, the experimental results in Figure 5.3 clearly state that edge nodes with high processing capabilities (i.e. lab4 and vrone with Octacore CPU and more than 4GB available RAM) ex-

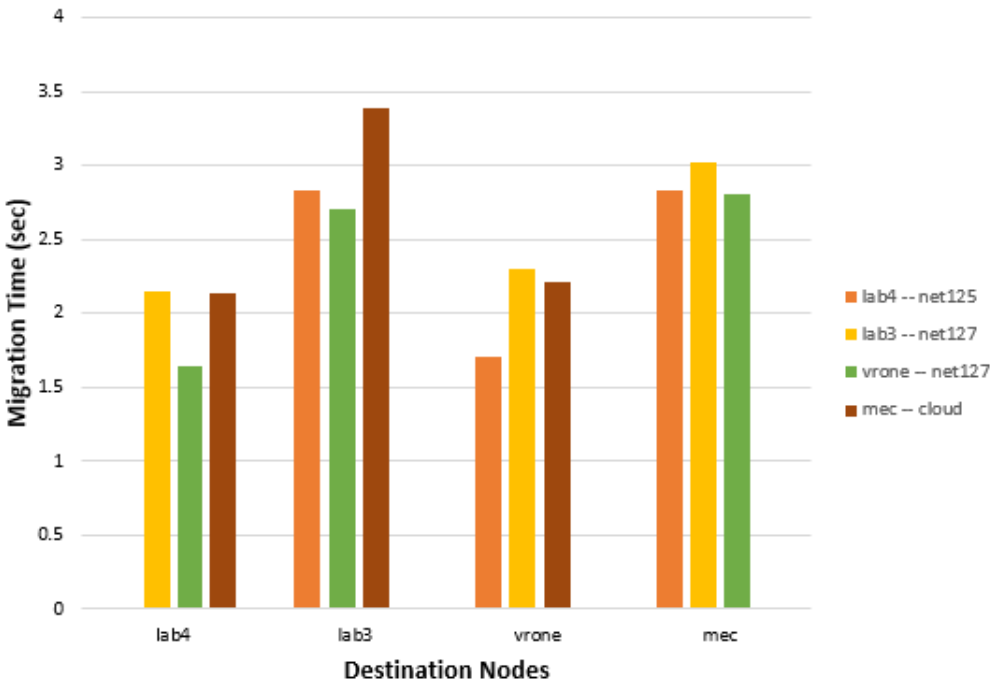

Figure 5.3: Stateful Migration Downtime

perience less service migration downtime as compared to an edge node with limited system resources (i.e. lab3 with 1GB available RAM and a Quadcore CPU). The overall migration downtime comprises of processing delay (on both source and destination to checkpoint and recreate the container) as well as network delay (to copy the memory contents to destination host). For vrone as the destination node, migration time from lab3 (i.e. from same network) is 2.3 seconds, which is 0.7 seconds higher than the migration time from lab4 (i.e. from different network). This shows that for limited capable edge node, the processing delay dominates the overall service migration downtime as compared to network delay. Due to persistent ssh-connection, migrating a container from cloud VM mec to vrone takes about 2.2 seconds which is 0.6 seconds more than the migration time from lab4 (i.e. 1.6 seconds). Now, in this case, 0.6 seconds correspond to additional network delay experienced when reaching a local edge node vrone from remote cloud device. For lab4 as destination, migration time from mec and and vrone almost remains the same as observed when vrone acted as destination.

Now when containers are migrated towards lab3, then it means that they must be checkpointed on source hosts and recreated on lab3. In this case, migration time becomes even much larger than the case when lab3 acted as the source node. From lab4 to lab3, a container migration takes about 1.2 seconds more time as compared to when container was migrated to vrone. Similarly, migrating a container from vrone takes almost 1 second more time compare to when migrating it towards lab4.

At last, the same process is repeated for mec. To migrate a container from lab4 to mec, it takes about 2.83 seconds, which is different and higher than the delay observed when migrating container from mec to lab4. Similar trend is observed when migrating from lab3 and lab4. This difference could be the result of different routes followed from edge nodes to remote cloud and vice versa.

5.2.2 Stateless Migration

In this type of migration, containers are suspended on source node and then restarted on destination from their initial memory state without requiring any need to copy the previous memory contents to destination. This significantly decreases the service downtime to almost 0.5 seconds as observed in Figure 5.4. In relation to stateful migration,

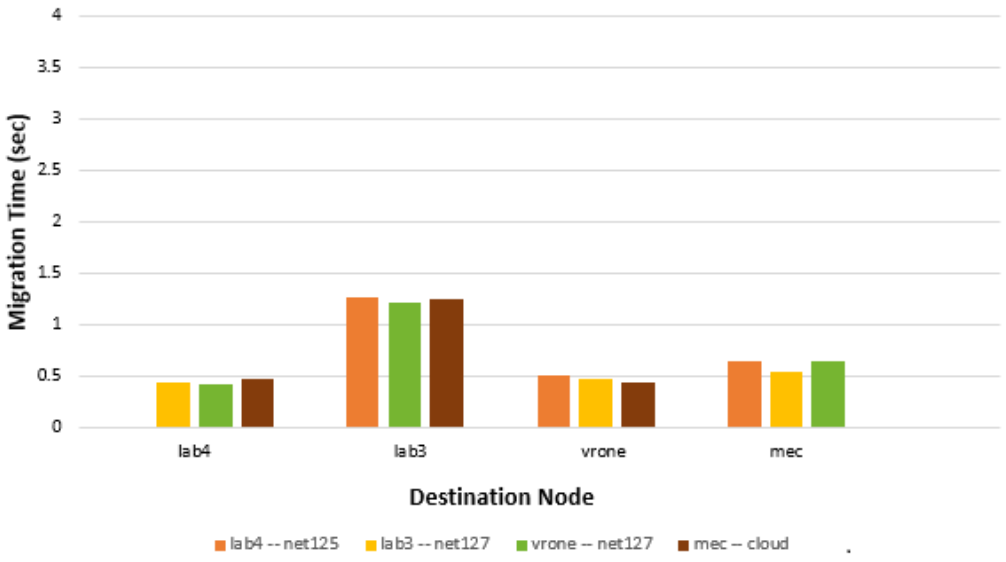

Figure 5.4: Stateless Migration Downtime

same experiments have been performed except that now containers are restarted from their initial start scripts and memory information. Except lab3, all other edge nodes require only 0.5 seconds to restart the container on destination. Similarly, for mec the stateless migration time is slightly higher (i.e. it ranges between 0.5-0.6 seconds).

5.3 Migration between Same Network Edge Nodes

To test the processing limits of Docker in terms of container migration, multiple containers are simultaneously migrated between edge nodes from same network. Figure 5.5 shows that with the introduction of each new service, the average migration delay increases. When three containers are migrated in parallel, then the average delay per each new container is almost 0.6 seconds. The same trend continues when four containers are migrated at same time instant. This corresponds to Docker limitation when multiple migration requests are received simultaneously, and Docker address them at cost of 0.6 seconds delay for each new service migration.

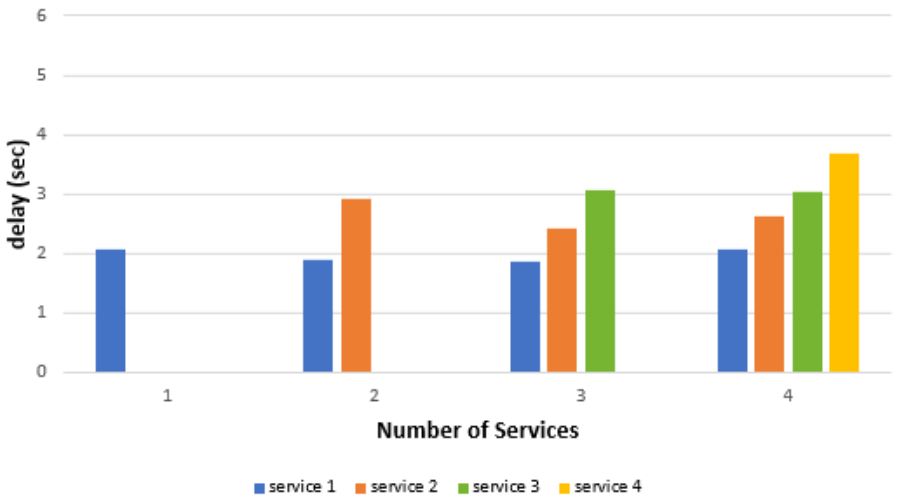

Figure 5.5: Migration between Same Network Edge Nodes

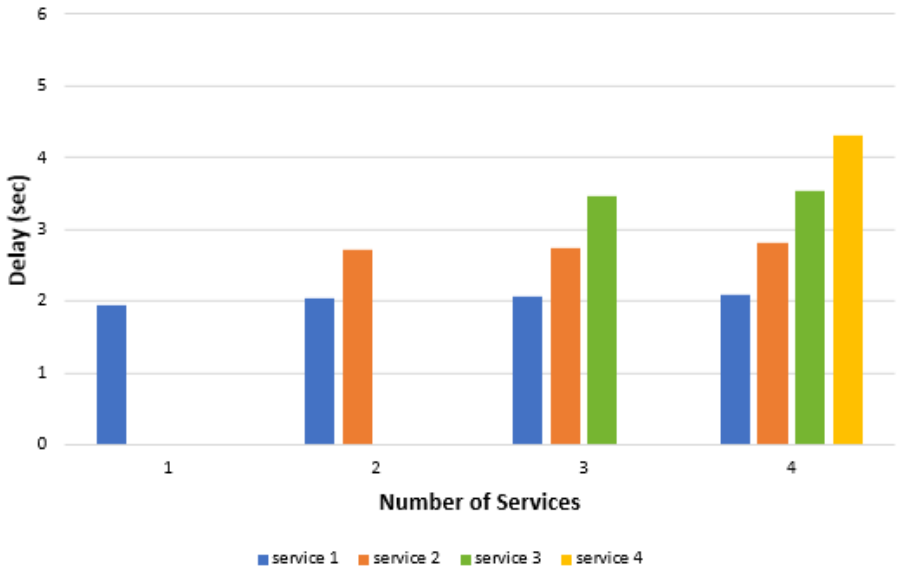

Figure 5.6: Migration between Different Network Edge Nodes

5.4 Migration between Different Network Edge Nodes

After getting an insight about service migration between same network edge nodes, now this experiment involves multiple service migration between different network edge nodes. As demonstrated in Figure 5.6, the obtained results depict a prominent increase in service downtime when they are migrated across different networks. Compared to previous experiment, the average downtime for each new service migration has increased to 0.7 seconds. However, the increment in total migration time show that each service migration suffers from the network delay of 0.3 seconds when migrating an application to the edge node from different network.

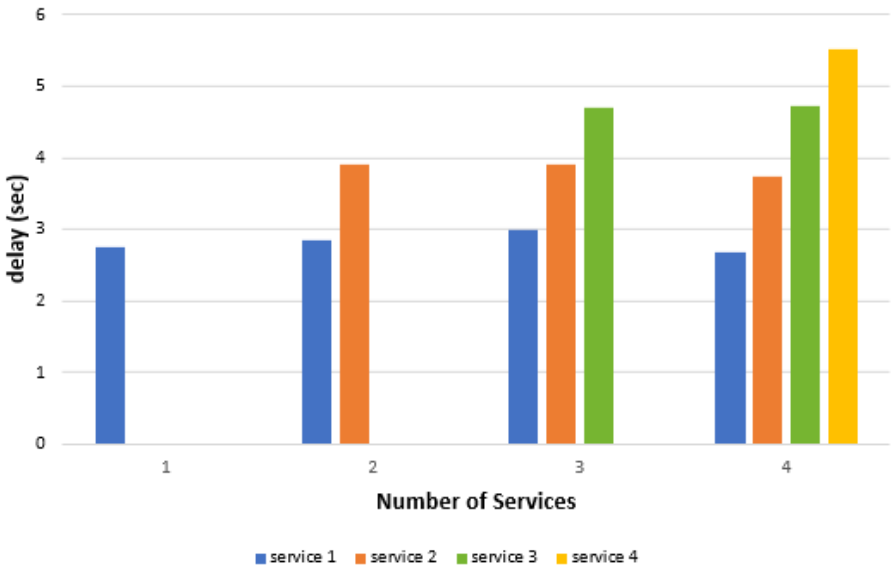

Figure 5.7: Migration From Edge Node to Cloud

5.5 Migration from Edge Node to Cloud

In this experiment, multiple containerized services are migrated from a local edge node to the remote cloud device as shown in Figure 5.7. The purpose is to understand the effect of network delay on service

migration when destination is at far off distance. Like previous experiment, same number of services are migrated in parallel to the cloud VM mec. Compared to migration between same network edge nodes, the total migration time has now increased beyond 1 second for each service. Similarly, the average downtime for each new service migration also becomes equal to 1 second. This shows that migration from an edge node to remote cloud entails unfavorable network delays which can limit the quality of service (QoS) and ultimately harm the user experience. However, service migration to cloud can avoid the outage issues when all edge node are unavailable or they lack sufficient resources for service provision.

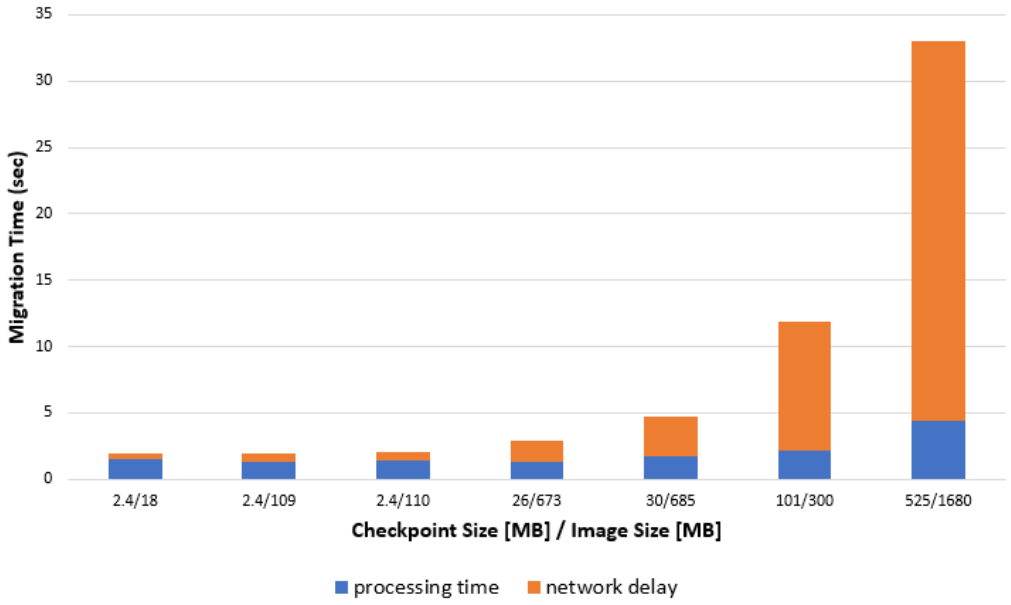

Figure 5.8: Effect of Checkpoint Size on Migration Downtime

5.6 Effect of Checkpoint Size on Migration Time

To explore the effect of checkpoint size (i.e. the payload to move towards the destination) on service migration downtime, containers of varying checkpoint sizes are migrated. From Figure 5.8, the observed results

clearly show that with the increase in checkpoint size, the corresponding service downtime increases exponentially. At the start, processing delay constitute almost 80% of overall service downtime, while network delay constitutes the remaining 20% delay. However, this trend changes significantly and ultimately get reversed when a container with large checkpoint size is migrated. This change occurs because copying the large files takes more time and hence the overall network delay is increased. On the other hand, the processing time (to checkpoint a container and then restart it from checkpoint) varies from 1.52 seconds to 4 seconds. This shows that for time-critical services, docker exceeds the minimum delay requirement and hence require an optimization mechanism (such as iterative pre-dump feature of CRIU or page-server model) to deliver near-zero downtime service migration.

5.7 Autonomous Migration

In all previous experiments, service migrations were triggered on behalf of user from control server dashboard. Now this experiment focuses on verifying the working feasibility of dynamic service migration using algorithm-I and II. On any edge node, these service migrations are triggered when high-load events are reported to control server. To emulate the high CPU load condition and fully utilize the CPU resources on any edge node, a Linux terminal command **yes > /dev/null &** is used. It′s one run execution utilizes about 25% CPU resources on lab3 with a quadcore processor. However, for edge nodes with an octa core processor, its one run iteration occupies about 10% of CPU resources. Multiple iterations of this command are used to generate the desired CPU load on any edge node. Now to test the system behavior with respect to algorithm-I and II, the following experiments are performed:

5.7.1 Service Migration Using Algorithm-I

To test the service migration using algorithm-I, three containers are deployed on lab3 with type defined as "stateful" and application profile named "nginx" having 10% CPU usage quota. Figure 5.9 shows that at this stage (i.e. scenario-I), the current CPU load on each node is less than 50%. Using one iteration of Linux stress tool on lab3, about 25% extra CPU load is injected. As a result, the CPU load on lab3 exceeds the normal CPU threshold (i.e. 50%). Now, as soon as this load-exceed

event is reported to server, it finds out all deployed services on lab3 and then finds out the appropriate destination node for each service. As algorithm-I gives priority to edge nodes from same network, so first container (app-1) is migrated to vrone because the aggregated value of app-1 required CPU and vrone current CPU level (i.e. 10+ 38 = 48) is less than 50%. However, for app-2, this value exceeds 50%, hence it is migrated to lab4. Similarly, the third container involves mismatch with lab4, so it is migrated to cloud VM (mec). The corresponding service migration downtimes exhibit the same behavior as observed in previous experiments.

Now, in scenario-II, two iterations of stress tool on lab4 incremented

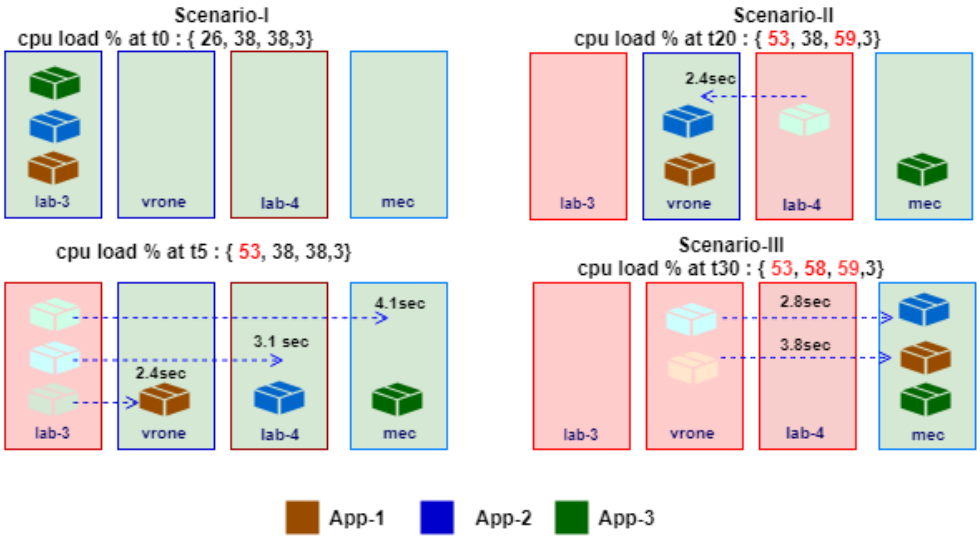

Figure 5.9: Service Migration Using Algorithm-I

the cpu load upto 59%. The rescue function on server detects the lab4 as red node and finds no green node from same network for app-2 migration. The only available green node is vrone from different network, so after matching application profile (i.e. 10+ 38 < 50), app-2 is migrated to vrone. For this migration, it only took about 2.4 seconds to migrate the app-2 from lab4 to vrone.

Finally, the scenario-III emulates the high-load situation on an edge node (i.e. vrone) when all other nearby edge nodes (i.e. lab4 and lab3) are unavailable. In this situation, algorithm-I migrates each application to cloud device at the same time instant. However, due to processing and network delays, each next service migration (app-1 at

present) took about 1 extra second to start on destination. These results clearly validate the service migration mechanism of algorithm-I and in real life, this can ensure service migration without causing congestion on destination host and hence enable load-balancing feature in edge cloudlets.

5.7.2 Service Migration Using Algorithm-II

Now this experiment evaluates the service migration mechanism of algorithm-II. The aim is to first select those edge nodes as destination which have highest system utilization and still they can accommodate affected applications of red nodes. To start the experiment, 4 containers are deployed on lab3 with each one requiring 10% CPU load. Using two iterations of stress tool, the CPU load on lab3 exceeds 50%. Figure

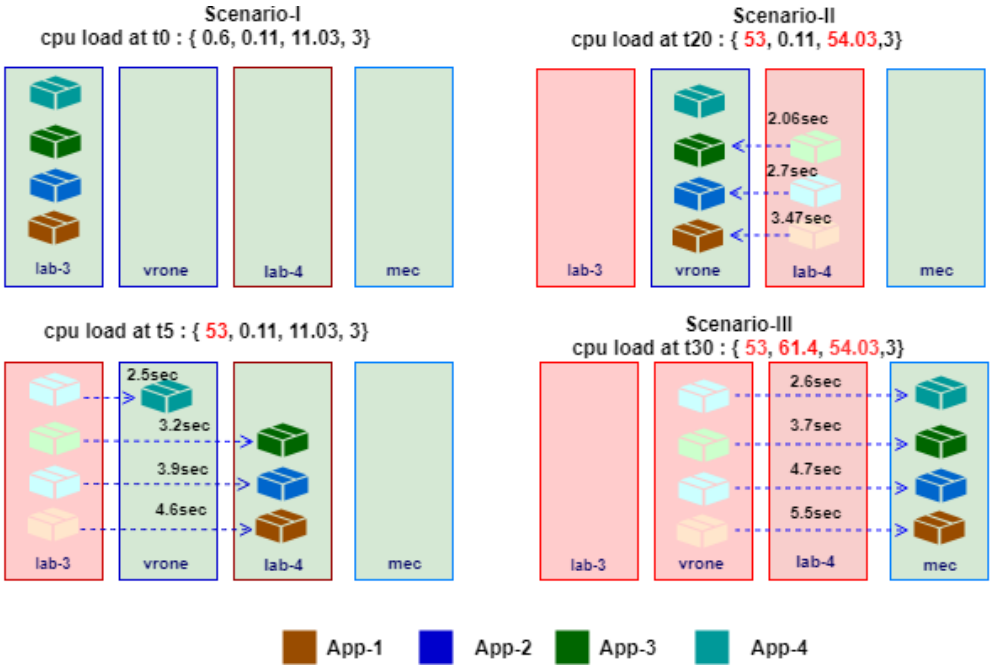

Figure 5.10: Service Migration Using Algorithm-II

5.10 show that at this stage, algorithm-II selects the highest utilized edge node (i.e. lab4) as the destination node for first three services (i.e. app-1, app-2 and app-3). After migrating three services, lab4 is unable to accommodate the fourth one and hence, it is migrated to next available highly utilized edge node (i.e. vrone). In scenario-II, high-load

emulation is performed on lab4 and as a result the deployed services are migrated to only available edge node (i.e. vrone). Like previous experiment, all services are successfully migrated to cloud when vrone also becomes a red node. In real network settings, when all services can be accommodated by a powerful edge node, we can switch the other available edge node into sleep mode to save some energy consumption and make it active again when needed.

Chapter 6

Conclusions

Emerging IoT and 5G technologies involve time-sensitive applications which require support for real-time low-latency communication, fast processing and data analytics. A new paradigm, known as Fog Computing (or MEC), has been introduced to meet these requirements. However, traditional Fog network architecture is composed of heavy-weight virtual machines, and since very few projects have explored OS-level virtualization in context of edge computing. Therefore, in this thesis work, container based MEC has been proposed. The developed control server also addresses the resource allocation challenge in Fog Computing. With a user-friendly dashboard, it provides the possibility to register the edge nodes, deploy and migrate the containerized services and monitor their health state information. Motivated by Follow Me Edge Cloud concept, two resource allocation algorithms have been developed for resource management and deciding on service migration when some edge node becomes overloaded. The developed testbed is then evaluated by conducting various experiments. The obtained results clearly show that event-driven strategy follows the low-latency requirement of edge computing. The container-based service provision is fast and easy to migrate. However, as service migration involves only pre-dump feature of CRIU, so all memory contents are copied to destination after freezing the container on source node. As a result, the service downtime is increased. Apart from it, docker-checkpoint on source and docker-start from checkpoint on destination takes almost 1.5 seconds which makes service processing delay unfavorable for latency-critical applications.

To improve the undesired processing time of Docker, it must be integrated with latest optimization tools such as page-server and iterative pre-dump feature of CRIU. At present, CRIU integration is only avail-

able for low-level containers (such as runc and lxc/lxd). However, some open-source projects (P.Haul, Flocker) are in active development phase, which will make Docker a feasible tool for real time container migration.

From design perspective, the developed platform is quite flexible that it can support new events and entities for desired integration. It is built using Docker′s Enterprise version 2.11 with built-in support for latest container orchestration tools such as Universal Control plane, Kubernetes and Docker Swarm. In future, this work can be extended to support 5G vehicular Fog Computing. We can register the mobile end-users and based on RSSI of nearby edge nodes, they can be provided a continued service when they switch the coverage area of one edge node to another. Furthermore, Fog Nodes can act as virtual traffic controllers when they process the real-time video streams of 5G connected cars and then, guide them in deciding major actions for autonomous driving such as lane change, speed control and car parking etc.

Bibliography

[1] About storage drivers. Tech. rep. https://docs.docker.com/storage/storagedriver/#container-and-layers. Accessed 14.7.2018.

[2] Advantages of containers, red hat do081x fundamentals of containers, kubernetes, and red hat openshift. Tech. rep. https://courses.edx.org/courses/course-v1:RedHat+DO081x+2T2017/course/. Accessed 9.7.2018.

[3] cadvisor. https://github.com/google/cadvisor. Accessed 07.08.2018.

[4] Checkpoint/restore. https://criu.org/Checkpoint/Restore. Accessed 30.7.2018.

[5] Comparison to other cr projects. https://criu.org/Comparison_to_other_CR_projects. Accessed 30.7.2018.

[6] container-migration/examples/migrateexample.sh. https://github.com/stmuraka/container-migration/blob/master/Examples/migrateExample.sh. Accessed 07.08.2018.

[7] Diskless migration. https://criu.org/Disk-less_migration. Accessed 30.7.2018.

[8] Docker overview. Tech. rep. https://docs.docker.com/engine/docker-overview/. Accessed 14.7.2018.

[9] Express tutorial part 4: Routes and controllers. https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/routes. Accessed 07.08.2018.

[10] Flocker. https://github.com/ClusterHQ/flocker. Accessed 30.7.2018.

[11] Fog computing. https://www.terminalworks.com/blog/post/2017/05/13/fog-computing. Accessed 31.7.2018.

[12] From checkpoint/restore to container migration. https://rhelblog.redhat.com/2016/09/26/from-checkpointrestore-to-container-migration/#more-2521. Accessed 30.7.2018.

[13] Fundamentals of containers, kubernetes, and red hat openshift: Docker core elements. https://courses.edx.org/courses/course-v1:RedHat+DO081x. Accessed 30.7.2018.

[14] How to integrate criu pre-dump feature with docker for iterative migration? https://github.com/checkpoint-restore/criu/issues/456. Accessed 30.7.2018.

[15] <iframe>: The inline frame element. https://developer.mozilla.org/en-US/docs/WenodeIPb/HTML/Element/iframe. Accessed 10.08.2018.

[16] Introduction to control groups (cgroups). Tech. rep. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/ch01. Accessed 9.7.2018.

[17] Isolate containers with a user namespace. Tech. rep. https://docs.docker.com/engine/security/userns-remap/. Accessed 9.7.2018.

[18] Live migration. https://criu.org/Live_migration. Accessed 30.7.2018.

[19] namespaces - overview of linux namespaces. Tech. rep. http://man7.org/linux/man-pages/man7/namespaces.7.html. Accessed 9.7.2018.

[20] On-demand vm provisioning for cloudlet-based cyber-foraging in resource-constrained environments. http://elijah.cs.cmu.edu./DOCS/echeverria-mobicase2014.pdf. Accessed 10.08.2018.

[21] Optimizing live container migration in lxd. https://lisas.de/~adrian/?p=1294. Accessed 30.7.2018.

[22] P.haul. https://criu.org/P.Haul. Accessed 30.7.2018.

[23] Understanding and securing linux namespaces. Tech. rep. https://www.linux.com/news/understanding-and-securing-linux-namespaces. Accessed 9.7.2018.

[24] Universal control plane overview. https://docs.docker.com/ee/ucp/. Accessed 30.7.2018.

[25] Weird ways of using application checkpoint/restore. https://www.youtube.com/watch?v=Ad2l4mctU-4&feature=youtu.be. Accessed 30.7.2018.

[26] What is kubernetes? https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. Accessed 30.7.2018.

[27] What socket.io is. https://socket.io/docs/#Using-with-Express. Accessed 07.08.2018.

[28] 2017), G. J. Gartner says 8.4 billion connected "things" will be in use in 2017, up 31 percent from 2016. Tech. rep., 2017. https://www.gartner.com/newsroom/id/3598917. Accessed 5.7.2018.

[29] ADDAD, R., DUTRA, D., BAGAA, M., TALEB, T., AND FLINCK, H. Towards a fast service migration in 5g, 06 2018.

[30] AND T. FUJII. Resource management for mobile edge computing using user mobility prediction. In *2018 International Conference on Information Networking (ICOIN)* (Jan 2018), pp. 718–720.

[31] B. CHUN, S.IHM, P. Clonecloud: Elastic execution between mobile device and cloud. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.226.1743&rep=rep1&type=pdf. Accessed 30.7.2018.

[32] CHAMOLA, V., THAM, C. K., AND CHALAPATHI, G. S. S. Latency aware mobile task assignment and load balancing for edge cloudlets. In *2017 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)* (March 2017), pp. 587–592.

[33] DUPONT, C., GIAFFREDA, R., AND CAPRA, L. Edge computing in iot context: Horizontal and vertical linux container migration. In *2017 Global Internet of Things Summit (GIoTS)* (June 2017), pp. 1–4.

[34] E.CUERVO, A.BALASUBRAMANIAN, D. A. S. R. P. B. Maui: Making smartphones last longer with code offload.

[35] INTERNETWORLDSTATS.COM. World internet users and 2018 population stats. Tech. rep., 2018. https://www.internetworldstats.com/stats.html. Accessed 5.7.2018.

[36] KOSTA, S., AUCINAS, A., HUI, P., MORTIER, R., AND ZHANG, X. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *2012 Proceedings IEEE INFOCOM* (March 2012), pp. 945–953.

[37] KUSEK, M. Internet of things: Today and tomorrow. In *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)* (May 2018), pp. 0335–0338.

[38] LAGWAL, M., AND BHARDWAJ, N. Load balancing in cloud computing using genetic algorithm. In *2017 International Conference on Intelligent Computing and Control Systems (ICICCS)* (June 2017), pp. 560–565.

[39] MACHEN, A., WANG, S., LEUNG, K. K., KO, B. J., AND SALONIDIS, T. Live service migration in mobile edge clouds. *IEEE Wireless Communications 25*, 1 (February 2018), 140–147.

[40] NADGOWDA, S., SUNEJA, S., BILA, N., AND ISCI, C. Voyager: Complete container state migration. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)* (June 2017), pp. 2137–2142.

[41] NAME, H. A. M., OLADIPO, F. O., AND ARIWA, E. User mobility and resource scheduling and management in fog computing to support iot devices. In *2017 Seventh International Conference on Innovative Computing Technology (INTECH)* (Aug 2017), pp. 191–196.

[42] QIU, Y., LUNG, C. H., AJILA, S., AND SRIVASTAVA, P. Lxc container migration in cloudlets under multipath tcp. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)* (July 2017), vol. 2, pp. 31–36.

[43] SATYANARAYANAN, M. Elijah: Cloudlet-based edge computing. `http://elijah.cs.cmu.edu/`. Accessed 30.7.2018.

[44] SINGH, M., AND AGRAWAL, R. Modified round robin algorithm (mrr). In *2017 IEEE International Conference on Power, Control, Signals and Instrumentation Engineering (ICPCSI)* (Sept 2017), pp. 2832–2839.

[45] YAO, D., GUI, L., HOU, F., SUN, F., MO, D., AND SHAN, H. Load balancing oriented computation offloading in mobile cloudlet. In *2017 IEEE 86th Vehicular Technology Conference (VTC-Fall)* (Sept 2017), pp. 1–6.

[46] ZHANG, H., ZHANG, Y., GU, Y., NIYATO, D., AND HAN, Z. A hierarchical game framework for resource management in fog computing. *IEEE Communications Magazine 55*, 8 (2017), 52–57.

[47] ZHU, C., PASTOR, G., XIAO, Y., LI, Y., AND YLAE-JAEAESKI, A. Fog following me: Latency and quality balanced task allocation in vehicular fog computing. In *2018 15th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)* (June 2018), pp. 1–9.